Étude théorique et implantation matérielle d'unités de calcul en représentation modulaire des nombres pour la cryptographie sur courbes elliptiques

Karim Bigou

ANNÉE 2014

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Karim Bigou

préparée à l'unité de recherche IRISA (UMR6074)
Institut de recherche en informatique et systèmes aléatoires
École Nationale Supérieure des Sciences Appliquées et de Technologie (ENSSAT) - Équipe CAIRN

Étude théorique et implantation matérielle d'unités de calcul en représentation modulaire des nombres pour la cryptographie sur courbes elliptiques

Thèse soutenue à Lannion le 3 Novembre 2014

devant le jury composé de :

Liam MARNANE
Senior Lecturer, University College Cork
rapporteur

Jean-Michel MULLER
Directeur de recherche CNRS, ENS Lyon, LIP
rapporteur

Jean-Claude BAJARD
Professeur, Université de Paris 6, LIP6
examinateur

Guy GOGNIAT
Professeur, Université de Bretagne Sud, Lab-STICC
examinateur

Arnaud TISSERAND
Chargé de recherche CNRS, IRISA
directeur de thèse

Nicolas GUILLERMIN
Expert cryptographie DGA
co-directeur de thèse

Remerciements

Je tiens tout d'abord à remercier mes directeurs de thèse, Arnaud Tisserand et Nicolas Guillermin, qui m'ont permis d'effectuer ce travail sur une problématique passionnante. Ils m'ont notamment mis le pied à l'étrier sur les aspects matériels, et Arnaud s'est montré extrêmement patient lorsqu'il a fallu rédiger les idées et les résultats pour les soumettre en conférence. Leurs précieux conseils m'ont permis d'aboutir au document que voici, dans les temps de mon financement de thèse (DGA-INRIA).

Je remercie aussi les autres membres du jury : Guy Gogniat en tant que président du jury, Jean-Claude Bajard en tant qu'examinateur, et enfin les relecteurs Liam Marnane et Jean-Michel Muller, qui sont venus de toute la France, et même d'Irlande.

Je tiens à remercier tous les membres de l'équipe CAIRN que j'ai pu côtoyer durant ma thèse. Cette équipe possède une formidable ambiance de travail, ce qui est très motivant et très encourageant dans les périodes difficiles qui juchent le chemin jusqu'à la soutenance de thèse. Je remercie plus particulièrement les secrétaires d'équipe Nadia et Angélique, mes anciens co-bureaux Thomas et Julien, les autres doctorants du groupe arithmétique/crypto Jérémy et Frank, et les doctorants qui ont démarré en même temps que moi, notamment Ganda-Stéphane et Hai.

Je remercie aussi tous ceux qui m'ont beaucoup soutenu durant la rédaction. En vrac, merci à Cédric, Christophe, aux trois Nicolas et à Mélanie. Les conseils de Nicolas E., les légendaires pestos de Nicolas V., les délicieuses mousses au chocolat de Mélanie, les discussions endiablées sur le jeu vidéo avec Nicolas S., le rendez-vous hebdomadaire *after work* à « l'Atmo » animé par Christophe et les combos de blagues réalisés par Cédric m'ont aidé à surmonter le rythme soutenu de fin de thèse.

Je remercie mes parents Gilles et Rachida et mes frères Nadjim et Jason pour tout leur soutien tout au long de ma thèse, et plus généralement, tout au long de ma scolarité. Je remercie mes parents d'avoir fait ces 20 heures de route en 3 jours pour pouvoir me soutenir encore une fois afin de conclure mes 8 années d'études.

Pour terminer, je remercie Julie, qui m'accompagne depuis bientôt 10 ans, et qui s'est investie dans cette thèse, notamment en relisant l'intégralité du document ici présent. Elle a dû faire face à mes périodes de doute, et m'a soutenu quelles que soient les humeurs qui m'ont traversées durant ces 3 dernières années. Elle a été mon soutien le plus fondamental durant toute la durée de ces travaux, et l'est encore aujourd'hui.

Table des matières

Liste des Algorithmes

Introduction

Contexte et motivations des travaux

La *sécurité* des *systèmes d'information* et de *communication* est, de nos jours, une question primordiale dans un bon nombre d'applications, et son champ d'application va avoir tendance à encore s'accroître avec l'utilisation et l'intégration de plus en plus massive de systèmes intelligents. Cette sécurité intervient dans des domaines différents comme la défense, l'économie numérique ou encore les divertissements, ce qui impose des contraintes très différentes. Le secret militaire, les transactions bancaires ou encore les services de vidéo à la demande sont des exemples que l'on peut associer à ces domaines. D'autres domaines apparaissent avec, par exemple, l'utilisation de circuits électroniques directement intégrés dans le corps à des fins médicales. Un autre exemple est la domotique, c'est-à-dire la construction de « maisons intelligentes » avec un contrôle informatique. On peut aussi évoquer des concepts plus généraux qui ont des besoins de sécurité, comme l'informatique dans le nuage (*cloud computing*). Le principe est d'utiliser la puissance de serveurs distants et leur capacité de stockage pour en profiter sur des dispositifs bien moins puissants, via le réseau (internet principalement). Une telle application génère beaucoup de trafic sur le réseau, et notamment beaucoup d'informations à protéger, avec potentiellement un gros débit d'informations à garder confidentielles, et beaucoup d'utilisateurs à authentifier. Une des promesses de ce type de service est de fournir autant de *confidentialité* sur les données stockées de façon distante que si elles étaient enregistrées localement.

La variété des contextes d'utilisation implique une variété de contraintes possibles : des contraintes de coût de mise en œuvre, de consommation d'énergie et de temps d'exécution ou de débit d'information doivent être prises en compte. Ces contraintes sont conditionnées par le *niveau de sécurité* que le concepteur cherche à obtenir ou à garantir. Par exemple, il n'est pas forcément nécessaire pour certaines applications d'avoir des protections qui résistent à des attaques de plusieurs années. Par contre, d'autres applications requièrent une protection pour des dizaines d'années, comme cela peut être le cas d'informations classées « Confidentiel Défense » ou « Secret Défense », qui sont prévues pour rester confidentielles 50 ans dans la loi.

Les systèmes d'information ayant des limites physiques, on ne peut pas obtenir une protection qui reste valable quelle que soit la puissance de calcul de l'attaquant. D'où une notion de niveau de sécurité, directement reliée à l'*attaque* la plus efficace que peut faire un attaquant, avec une certaine quantité de ressources. La sécurité repose sur différents types de *protocoles*, qui permettent par exemple d'authentifier les interlocuteurs et de garder secrète leurs communications. Ces protocoles utilisent des primitives cryptographiques pour assurer ces fonctions. La cryptographie classique, utilisée actuellement dans nos systèmes de communication, repose sur des *problèmes mathématiques* qui sont *a priori impossibles*

à résoudre à un coût raisonnable lorsqu'on ne possède pas un certain *secret* : *la clé*. La difficulté de ces problèmes est choisie pour être adaptée à la protection que l'on veut mettre en place et contre quel attaquant on veut se protéger : un particulier, une entreprise ou un état par exemple. Ces paramètres font que la mise en place de protections adaptées est un problème complexe.

Il existe plusieurs types de cryptographie. Actuellement, dans les protocoles de sécurité, on utilise généralement deux types de cryptographie : la *cryptographie asymétrique* (ou à clé publique) et la *cryptographie symétrique* (ou à clé secrète). Ces deux catégories ont des caractéristiques différentes et sont utilisées généralement de façon complémentaire. La cryptographie à clé publique ne requiert pas de secret partagé entre les deux interlocuteurs, celui qui reçoit le message possède 2 clés, une publique qu'il diffuse, et un autre qu'il garde secrète. Ce type de cryptographie requiert des calculs compliqués sur des grands nombres, ce qui rend ces opérations coûteuses en temps. Les travaux de cette thèse s'intéresse exclusivement à ce type de cryptographie. Plus précisément, on va s'intéresser à l'accélération des calculs pour la *cryptographie sur courbes elliptiques* (ECC pour *elliptic curve cryptography*), qui a été introduite dans le milieu des années 1980 par Koblitz [67] et Miller [80]. D'autres primitives de cryptographie asymétrique seront plus brièvement étudiées, comme RSA [102] (du nom de ses auteurs Rivest, Shamir et Adleman) et Diffie-Hellman [38]. De l'autre côté, la cryptographie symétrique a comme contrainte forte que les deux interlocuteurs doivent partager une même clé secrète. Grâce à ce secret partagé, ils vont pouvoir utiliser un des algorithmes de chiffrement symétrique comme l'AES [90] par exemple, qui est un chiffrement symétrique très rapide. La complémentarité des deux types de cryptographie se dessine lorsque deux interlocuteurs cherchent à établir une communication via un canal sécurisé. Ces interlocuteurs devront d'abord s'authentifier mutuellement puis échanger une clé secrète grâce à la cryptographie asymétrique. Ensuite, ils pourront communiquer de manière sécurisée en utilisant un chiffrement symétrique avec la clé qu'ils ont partagée, de manière bien plus rapide que si tout était fait avec la cryptographie asymétrique.

Il existe aussi d'autres propositions en terme de cryptographie, mais qui ne sont pas utilisées car trop peu performantes pour des besoins actuels. Par exemple, la distribution de clé quantique est une méthode qui est basée sur des phénomènes physiques, à la différence de la cryptographie actuelle basée sur certains problèmes mathématiques. L'avantage principal de ce type de cryptographie est qu'elle est résistante aux attaques que l'on pourrait implanter si un véritable ordinateur quantique était mis au point un jour. Un exemple célèbre est l'algorithme de Shor [113] permettant de factoriser des entiers très rapidement sur un ordinateur quantique. Cette factorisation permettrait par exemple de casser très rapidement RSA. Actuellement, le plus grand nombre entier décomposé en ses facteurs premiers par un calculateur quantique est 21 [75]. L'utilisation massive, un jour, de ce type de cryptographie est extrêmement hypothétique, et est largement en dehors du cadre de cette thèse.

Cette thèse porte principalement sur l'*accélération* de certaines opérations sur des nombres de plusieurs centaines à plusieurs milliers de bits pour les calculs nécessaires à la cryptographie asymétrique, en particulier ECC. Les courbes elliptiques sont très intéressantes parmi l'ensemble des primitives de cryptographie asymétrique car elles permettent d'utiliser des tailles de clé plus petites que les autres standards, pour une sécurité équivalente [57]. Il en va de même pour la taille des nombres traités et les performances en temps

d'exécution. Par exemple, ECC définit avec des clés de 160 bits et des nombres représentés sur 160 bits, propose une *sécurité équivalente* à RSA sur 1024 bits (taille des clés et des éléments). Plus généralement, les tailles des valeurs pour ECC vont de 160 à 600 bits et pour RSA de 1024 à 3072 dans les standards. Les tables 1.1, 1.2, 1.3 et 1.4 aux pages 32 et 33 présentent des comparaisons sur la taille des clés, la vitesse d'exécution ou encore la consommation d'énergie.

On trouve deux grandes catégories de courbes elliptiques dans l'état de l'art d'ECC : les courbes ayant comme *corps de base* $\mathbb{F}_{2^m}$ et celles définies sur $\mathbb{F}_p$. Les courbes définies sur $\mathbb{F}_{2^m}$ proposent une arithmétique généralement plus efficace, car le corps de base $\mathbb{F}_{2^m}$ possède plus de structure permettant l'accélération des calculs. Par exemple, on a $(x+y)^2 = x^2 + y^2$ dans $\mathbb{F}_{2^m}$, ce qui permet d'accélérer certains calculs. Cependant, on ne sait pas si cette structure sur le corps de base peut aider un attaquant et réduit la sécurité du système. Les travaux de cette thèse porteront exclusivement sur ECC sur $\mathbb{F}_p$. Ces courbes sont, par exemple, recommandées par l'agence américaine NSA (*national security agency*) pour l'*échange de clé* et la *signature numérique* des documents classés « *Secret* » et « *Top Secret* » avec des corps de 256 bits et 384 bits (voir [88, 89]). Les deux types de courbes sont intéressants car, premièrement, il est important en cryptographie de pouvoir disposer de différentes protections pour pallier l'éventuelle défaillance de l'une d'entre elles, et deuxièmement, il existe un bon nombre de brevets sur l'implantation efficace des 2 types de courbes qui peuvent guider le choix du type de corps ($\mathbb{F}_{2^m}$ ou $\mathbb{F}_p$).

Si les courbes elliptiques ont de si petites clés et de si petites tailles d'éléments par rapport à RSA et Diffie-Hellman, c'est grâce à leur meilleure résistance aux *attaques mathématiques connues*. Le champ de la cryptologie s'intéressant aux attaques mathématiques se nomme la *cryptanalyse*. Par exemple, c'est en partie parce que nous ne savons pas mettre en œuvre l'attaque du calcul d'indices [3] sur les courbes elliptiques qu'elles utilisent des entiers de centaines de bits, comparés aux milliers de bits pour RSA et Diffie-Hellman. Pour illustrer cette différence, on peut comparer les records des tailles maximales qui ont été cassées pour RSA et pour ECC dans la littérature. Ainsi, un module RSA de 768 bits a été factorisé dans [65] fin 2009, alors que le record pour ECC est seulement de 109 bits, pour les 2 types de corps $\mathbb{F}_{2^m}$ et $\mathbb{F}_p$ (2002 et 2004, voir [24]). Le prochain niveau du concours pour ECC est fixé à 131 bits par Certicom, qui est l'entreprise organisatrice du concours et détentrice de centaines de brevets sur les implantations ECC. Du côté de RSA, des équipes de recherche travaillent actuellement à la résolution de RSA sur 1024 bits.

Malgré les bonnes propriétés mathématiques d'ECC, d'autres types d'attaques existent utilisant les fuites d'information des implantations, comme la consommation d'énergie [69] ou le rayonnement électromagnétique [4]. C'est d'autant plus vrai lorsque nous travaillons sur des implantations sur systèmes embarqués, où des protections spéciales doivent être mises en œuvre. On appelle celles-ci *attaques par canaux cachés/auxiliaires* (SCA pour *side channel attack*). Les travaux de la thèse sont effectués dans un contexte d'implantation circuit/systèmes embarqués, ce type d'attaques est donc à prendre en compte. L'objectif est d'avoir une implantation rapide et sûre. Nos travaux portent sur l'accélération des calculs ECC, tout en étant compatibles avec les protections habituelles de l'état de l'art.

La taille des nombres avec lesquels on calcule en cryptographie asymétrique (160–521 bits pour ECC, 1024–3072 pour RSA/Diffie-Hellman) donne une importance capitale à l'arithmétique implantée pour effectuer les différentes opérations élémentaires. Ce qu'on

entend ici par opérations élémentaires sont les additions/soustractions, les multiplications et les réductions modulaires dans le corps de base. Les implications des choix d'implantation de ces opérations sont diverses. De très gros opérateurs permettront de gagner un ordre de grandeur en temps d'exécution par exemple, mais consommeront plus d'énergie et peuvent être ainsi plus visibles sur une attaque d'analyse de consommation d'énergie. D'un autre côté, si la priorité n'est pas le *temps d'exécution* mais la *consommation d'énergie* ou la *surface de silicium* utilisée, un petit opérateur, lent, peut suffire. Plus généralement, les contraintes typiques que l'on retrouve pour de tels opérateurs sont la surface utilisée, que l'on peut relier aux moyens financiers nécessaires pour concevoir et faire le circuit, la vitesse d'exécution, le débit de traitement des données et la consommation d'énergie.

Pour réaliser des opérateurs arithmétiques répondant aux contraintes fixées par l'application, diverses possibilités existent. Premièrement, on peut toujours doubler le débit si on s'autorise à doubler la surface de circuit, en utilisant deux instances du même opérateur. On peut aussi augmenter le nombre d'étages de *pipeline*, ce qui peut accroître la fréquence d'utilisation et le débit de données mais, en contrepartie, augmente la surface et la consommation d'énergie. Les *choix arithmétiques* influent aussi sur les *performances* de l'opérateur. Par exemple, la *propagation de retenue* dans l'addition ou la multiplication sont un frein à la *parallélisation* des opérateurs arithmétiques sans représentation adaptée. Certaines représentations des nombres permettent tout de même de paralléliser l'addition ou même la multiplication. C'est le cas de la *représentation modulaire des nombres RNS* (pour *residue number system*) qui est l'objet de cette thèse. Plus généralement, la *représentation des nombres* dans une implantation de cryptographie asymétrique est un élément important à prendre en compte pour accélérer les calculs. Par exemple, certaines représentations, comme RNS, permettent de faire les additions et les multiplications en parallèle. De plus, à chaque représentation correspond des coûts de calcul différents pour les opérations élémentaires addition, multiplication et réduction modulaire, ce qui peut encourager son utilisation pour certaines applications. Certaines opérations spécifiques sont même extrêmement efficaces, et peuvent mener à repenser certains algorithmes. Par exemple, en numération simple de position en base 2, la division et la multiplication par 2 sont très efficaces : elles correspondent à de simples décalages.

Le standard américain du NIST [91] propose l'utilisation de courbes elliptiques définies sur des corps qui permettent de tirer parti de la représentation en base 2. Il est proposé d'utiliser des nombres premiers très particuliers, les pseudo-Mersenne [79], permettant d'effectuer les réductions modulaires très efficacement lorsque les valeurs sont représentées en base 2. Par exemple, un des corps proposé est $\mathbb{F}_{P_{521}}$ avec $P_{521} = 2^{521} - 1$, pour lequel 2 additions suffisent pour effectuer une réduction modulaire. Nous verrons dans les travaux de cette thèse qu'il est possible de trouver des nombres premiers qui sont eux adaptés à la représentation modulaire des nombres, que nous allons brièvement introduire ci-dessous.

La représentation modulaire des nombres (RNS)

Dans cette thèse, une segmentation va être faite entre la numération simple de position qui est de très bas niveau et la représentation finale des valeurs. Nous allons toujours représenter les nombres avec des « groupes de bits », mais organisés de façon bien différente de la représentation habituelle. Cette représentation, appelée représentation modulaire des nombres ou RNS [49, 119] (pour *residue number system*) a été proposée à l'origine pour des applications de traitement du signal [26, 60] et plus récemment pour des calculs cryp-

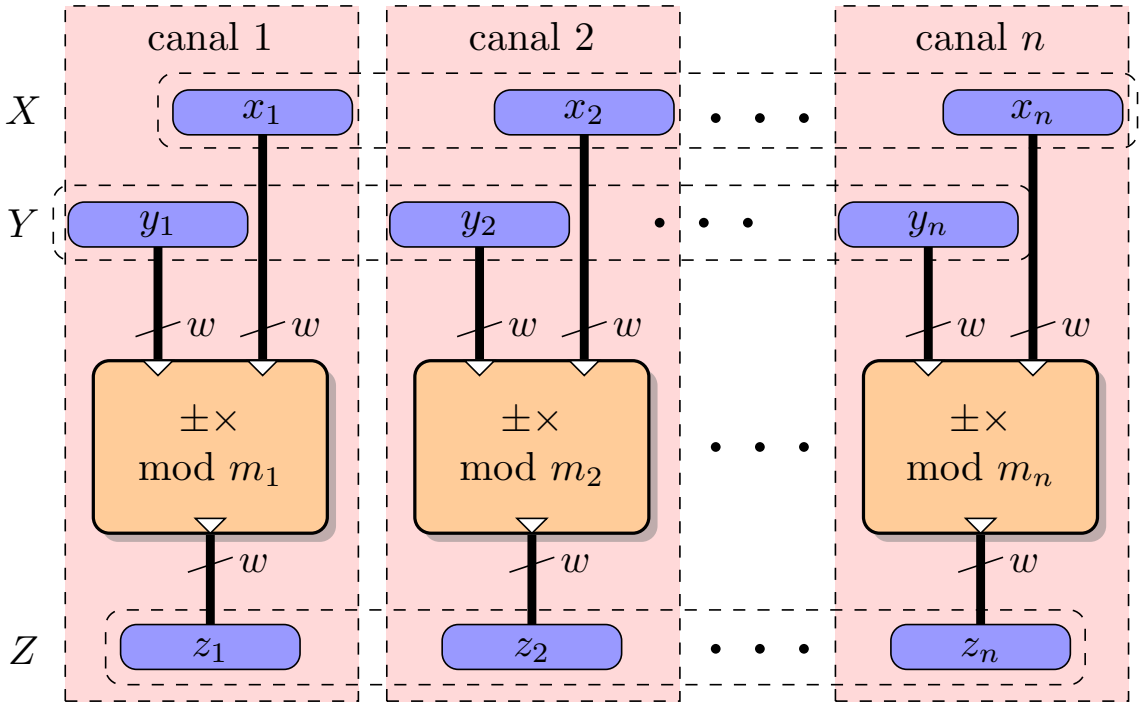

FIGURE 1 – Addition et multiplication sur les canaux parallèles en RNS.

tographiques [10, 52, 87]. Cette représentation permet d'effectuer des calculs rapides sur de très grands éléments, entiers ou éléments de grands corps finis, ce qui est très intéressant pour la cryptographie asymétrique. Dans cette représentation, les entiers de ℓ bits sont découpés en n morceaux bien plus petits, de w bits seulement. Typiquement, pour ECC, on a $\ell \in [160, 600]$ (pour les besoins de sécurité actuels) et $w \in [16, 64]$ en pratique sur les circuits actuels, avec $n \times w \geqslant \ell$. En RNS, un entier X sera représenté par $\overrightarrow{X} = (x_1, \ldots, x_n) = (X \bmod m_1, \ldots, X \bmod m_n)$, où $(m_1, \ldots, m_n)$ sont des *moduli* de w bits. On dit alors que $(m_1, \ldots, m_n)$ est la *base RNS* et $\overrightarrow{X}$ la représentation RNS de X. On appelle *canal* tous les traitements effectués modulo un des éléments de la base. L'originalité du RNS vient du fait que pour un certain nombre d'opérations, dont la multiplication et l'addition, les calculs sont fait indépendamment sur chacun des morceaux. En fait, on a :

$$\overrightarrow{X} \diamond \overrightarrow{Y} \;=\; (|x_1 \diamond y_1|_{m_1}, \ldots, |x_n \diamond y_n|_{m_n}) \quad ,$$

avec $\diamond \in \{+, -, \times\}$. Les multiplications et les additions/soustractions sont donc découpées en petites opérations *indépendantes* sur w bits. Ces petites opérations sont *parallèles* de façon naturelle, permettant d'obtenir très rapidement le résultat d'une somme ou d'un produit. La figure 1 illustre le parallélisme du RNS et l'indépendance des calculs sur les différents moduli/canaux. On a deux arithmétiques distinctes : l'arithmétique sur chacun des canaux et l'arithmétique combinant les canaux pour faire les opérations sur la totalité de l'entier qu'on appellera arithmétique RNS. L'arithmétique des canaux, c'est-à-dire modulo l'un des m_i, est en fait l'*arithmétique modulaire* classique, sur w bits. On représente les valeurs en numération simple de position et on choisit les moduli pour que les calculs soient efficaces, en choisissant des pseudo-Mersenne. Un travail très récent déroge au choix des pseudo-Mersenne, proposé par Bajard et Merkiche [14], ce qui permet d'avoir plus de choix pour définir la base RNS. Ils effectuent des petites réductions de Montgomery [82]

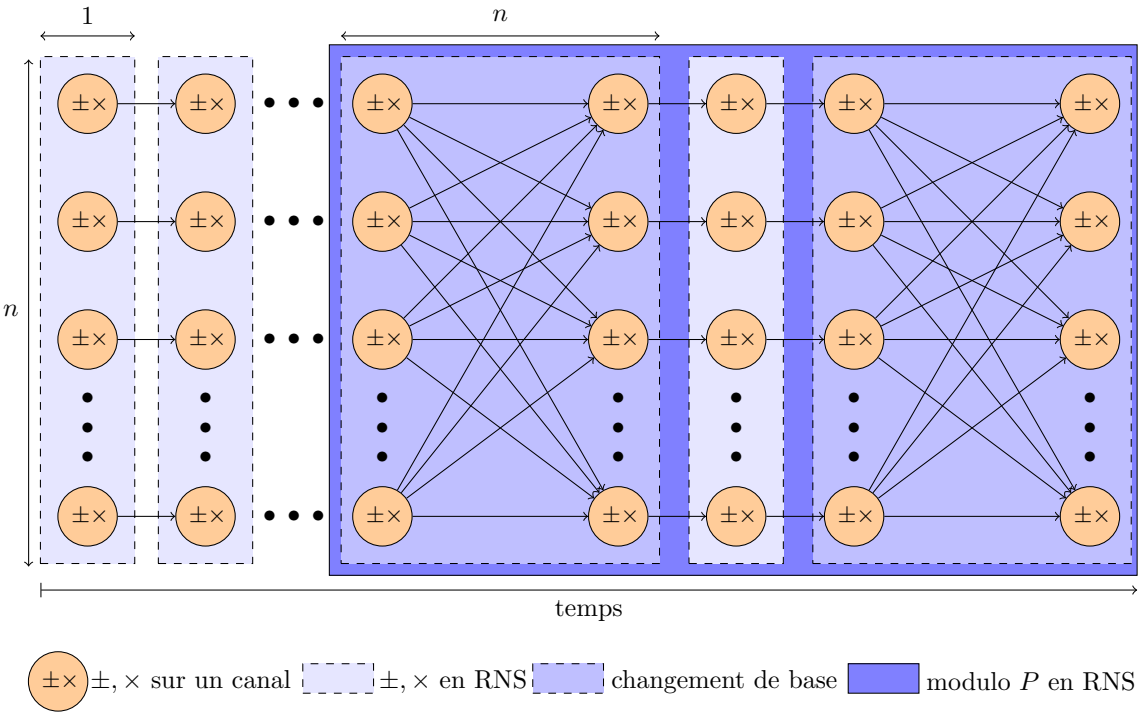

FIGURE 2 – Séquence de calcul typique en RNS pour la cryptographie : enchaînement de multiplications et d'additions RNS suivies d'une réduction modulo P en supposant les opérations sur les canaux parallélisées.

sur les canaux de w bits. Les algorithmes de cette thèse porteront eux sur l'arithmétique RNS, c'est-à-dire sur la combinaison des valeurs dans les différents moduli pour effectuer nos calculs efficacement pour des opérations plus compliquées que la multiplication, comme la réduction modulaire et l'inversion modulaire.

Au vu de la figure 1 seule, on pourrait croire qu'avec n *unités arithmétiques*, il suffit de prendre des moduli très petits, c.-à-d. avec w très petit, pour calculer le plus rapidement possible. Il est vrai qu'en réduisant w, on réduit la complexité des calculs sur chacun des canaux. Par contre, on va augmenter celle de la *réduction modulaire*, qui est une opération essentielle aux calculs cryptographiques. En effet, la multiplication et la réduction modulaire sont en contradiction en RNS pour le choix de la taille des moduli. La figure 2 illustre le coût d'une réduction modulaire et d'une multiplication ou addition RNS suivant n, le nombre de moduli. Alors qu'une multiplication ne demande que n multiplications de w bits, qui peuvent toutes être effectuées en parallèle, la réduction modulaire a un coût quadratique ($\approx 2\,n^2$). En supposant que nous puissions effectuer n multiplications ou additions en parallèle sur une architecture, une multiplication sera effectuée en à peu près $2n$ fois moins de cycles que la réduction modulaire. Ceci est dû à l'utilisation de *changements de base* [120] pour la réduction modulaire, qui est une fonction permettant de passer d'une représentation RNS définie par une première base, à une seconde représentation RNS définie par une deuxième base. Cette fonction est, actuellement, indispensable pour faire une réduction modulaire. Le côté quadratique du changement de base est illustré à la figure 2 : chacun des n moduli de la base d'arrivée va opérer sur les valeurs correspondant aux n moduli de la base de départ, ce qui donne bien n^2 opérations. Pour chaque moduli, n opé-

rations sont effectuées, nous n'avons pas déroulé ces opérations dans la figure 2 par soucis de place. Comme nous le verrons dans le chapitre 1, le choix du nombre de canaux n'est pas une question facile, et n'a pas été très étudié pour des tailles cryptographiques.

Une des subtilités de la représentation RNS est que nous devrions plutôt parler de représentations au pluriel. En effet, à la différence de la numération simple de position en base 2 qui est un objet très précis, une représentation RNS va être définie par un ensemble de moduli. C'est d'ailleurs ce qui d'une part, apporte la flexibilité de la représentation, et d'autre part, permet de trouver de nouvelles optimisations en jouant sur l'interaction entre la définition des moduli et l'arithmétique RNS. La plupart des algorithmes de l'état de l'art ne posent que de faibles contraintes sur le choix des moduli, généralement pour garantir une bonne arithmétique sur ceux-ci (on choisit des nombres pseudo-Mersenne). C'est un peu comme si on proposait des algorithmes pour la numération simple de position sans utiliser les spécificités de la base 2. Certaines propositions ont quand même été faites en contraignant plus fortement le choix des moduli, afin d'améliorer l'arithmétique RNS et seront présentées dans l'état de l'art. Par contre, le fait de ne pas trop contraindre le choix des moduli permet d'utiliser certaines protections contre les attaques par canaux cachés basées sur le RNS, comme la contre-mesure *Leak Resistant Arithmetic* (LRA [11]). Cette protection consiste à tirer au *hasard* une base avant d'effectuer le calcul cryptographique, ce qui a pour effet de changer la représentation des éléments, et permet de lutter contre des attaques statistiques sur plusieurs exécutions de ce calcul.

Dans le cadre d'implantations matérielles pour la cryptographie, la représentation RNS n'a pas encore connu beaucoup d'évolutions. En effet, la plupart des implantations sont complètement parallélisées, c'est-à-dire avec autant d'unités de calcul que de canaux, afin d'obtenir un temps d'exécution le plus faible possible (cf. table 1.6 dans la section 1.3.5). Parfois, pour certaines implantations sur de grands paramètres, comme pour RSA, un grand diviseur du nombre de moduli a été choisi. On peut citer en exemple l'implantation de Nozaki *et al.* [87], qui intègre 11 unités arithmétiques parallèles, pour des bases de 22, 33 ou 66 moduli. Pourtant, la flexibilité du RNS devrait mener à toute une zoologie de types d'implantations différentes, que ce soit grâce à l'indépendance des canaux ou encore la flexibilité sur la paramétrisation des canaux. En effet, la modularité que permet naturellement la représentation peut sembler sous-exploitée au vue des références de l'état de l'art. En réalité, comme présenté au chapitre 1, le nombre de propositions est en train d'augmenter, bien que souvent basées sur le même modèle. La modularité a au moins deux avantages pour nos applications. Premièrement, elle permet de changer facilement la surface allouée à notre circuit, sans changer les opérateurs implantés : il suffit de retirer ou d'ajouter des unités arithmétiques de calcul sur les moduli. C'est un véritable atout lorsque l'on considère la conception d'un produit pour qui la cryptographie et la sécurité ne sont seulement que des fonctionnalités annexes, certes nécessaires, du cahier des charges, comme par exemple pour un service de vidéo à la demande. Deuxièmement, grâce à cette modularité, on peut de réduire le nombre d'unités arithmétiques, permettant de rendre aléatoire l'ordre des moduli sur lesquels on calcule, à l'intérieur d'une même base, pour protéger le circuit contre certaines attaques par canaux cachés.

Outre une partie des possibilités d'implantation qui n'ont pas été exploitées, la représentation elle-même n'a subi que très peu de modifications pour fournir de meilleurs résultats. Les mises en œuvre du RNS ne s'éloignent finalement que très peu de la définition provenant du *théorème des restes chinois*. Par opposition, l'arithmétique sur la représentation

classique binaire a subit toute sorte de modifications pour accélérer les calculs, que ce soit pour les additions ou les multiplications (par exemple en introduisant de la redondance dans la représentation). Dans le cas du RNS, le calcul modulaire utilise 2 bases de tailles égales depuis que les premières adaptations RNS de l'algorithme de réduction modulaire de Montgomery ont été proposées [6, 64, 99]. Les améliorations de la réduction modulaire en RNS sont depuis lors toujours restées sur cette approche, généralement en améliorant des sous-parties de l'algorithme. La représentation n'a été véritablement modifiée que récemment, avec les travaux de Gandino *et al.* [48]. Les auteurs se sont autorisés à ne pas travailler exactement en représentation RNS dans la seconde base, mais cette modification leur a permis de réduire le nombre de calculs car celle-ci s'intègre parfaitement dans l'algorithme de réduction de Montgomery RNS.

Même si dans les détails la proposition de Gandino *et al.* [48] reste très proche du RNS habituel, elle marque peut-être le début d'une évolution beaucoup plus profonde de l'arithmétique RNS. Le RNS est utilisé pour faire le lien entre les mathématiques et l'arithmétique d'un côté, et le matériel de l'autre : il y a donc deux directions vers lesquelles tendre pour améliorer les implantations. Le travail de Gandino *et al.* porte essentiellement sur la réduction du nombre d'opérations, leur travail tend donc à améliorer le côté arithmétique même si cela permet au final de réduire le nombre de cycles dans leur implantation RSA en RNS. Elle ne porte pas sur une évolution du RNS pour le matériel. Il n'y a pas encore eu, à notre connaissance, de véritables modifications de la représentation afin d'obtenir une architecture moins coûteuse par exemple. Il est quand même souvent considéré que w, la taille des moduli, est très proche de la taille des mots machines, ou des multiplieurs des blocs DSP pour les implantations FPGA. Malgré tout, cette considération est juste un choix plus ou moins astucieux des moduli, et non pas une transformation de la représentation pour le matériel. On essaye généralement de faire une bonne implantation du RNS classique plutôt que d'essayer de modifier celui-ci pour avoir une bonne implantation. Après tout, la représentation RNS hérite naturellement d'un certain nombre de bonnes propriétés pour le matériel, comme la non propagation de retenue pour l'addition et la multiplication, et son haut niveau de parallélisme. C'est donc déjà un bon point de départ pour le matériel, mais ce potentiel là n'a peut-être pas, et même sûrement pas, atteint son maximum. Dans un futur proche, outre les améliorations arithmétiques, la représentation RNS gagnera sûrement aussi sur son implantation.

Pour finir, les caractéristiques particulières du RNS permettent non seulement d'obtenir des implantations cryptographiques rapides, mais aussi de protéger le circuit. Nous avons évoqué l'existence de propositions contre les attaques statistiques sur la consommation de courant ou le rayonnement électromagnétique avec le tirage aléatoire de l'ordre des calculs ou de la base (protection LRA [11]). Il existe aussi des protections contre les attaques par injection de faute dans [9] et [53]. Des premières études d'implantations de telles protections ont récemment été publiées, par exemple [94], aboutissant à de bons résultats en terme de protection. Ces implantations sont par contre toujours proposées dans une architecture complètement parallèle, afin de se comparer à des résultats d'implantation sans protection en RNS. On pourrait aussi imaginer des architectures beaucoup plus orientées sécurité que performance, et utiliser la modularité que propose la représentation RNS pour avoir encore plus d'aléa que dans le cas complètement parallèle. C'est aussi un aspect important du RNS, mais qui n'a pas été traité dans cette thèse pour des raisons de temps.

Les sujets de recherche sur la représentation RNS sont donc assez vastes, et beaucoup

de travaux seront encore publiés sur le sujet. Pour conclure cette introduction, les travaux de cette thèse vont être rapidement introduit via la présentation du plan de la thèse.

Objectifs et plan de la thèse

Les objectifs de la thèse sont d'améliorer les performances des implantations cryptographiques utilisant le RNS, avec comme objectif particulier ECC. Ces améliorations doivent être raisonnables en terme de surface de circuit utilisée, nous avons donc gardé comme support architectural l'implantation proposée par Guillermin dans [52], qui est l'état de l'art des implantations ECC en RNS, et est elle-même basée sur l'architecture de Kawamura *et al.* [64]. Ces contributions gardent les bonnes propriétés du RNS (indépendance des canaux), mais vont parfois utiliser certaines contraintes pour plus d'efficacité.

Le premier chapitre porte sur le contexte de mes travaux, et l'état de l'art dans lequel ils s'inscrivent. La cryptographie asymétrique y est présentée plus en détail, en introduisant les cryptosystèmes qui vont nous intéresser tout au long du document. Certaines techniques classiques de calcul pour la cryptographie seront présentées, ainsi que certaines notions sur la sécurité des implantations de celles-ci. Ensuite sont présentés certains algorithmes de l'arithmétique modulaire classique, en numération simple de position en base 2. Ceux-ci sont souvent à la base des algorithmes de l'état de l'art en RNS (ou du moins leurs idées fondatrices), et le travail présenté au chapitre 2 reprend par exemple des idées sur les algorithmes d'inversion. Enfin, le chapitre 1 se termine sur un état de l'art de l'arithmétique RNS pour la cryptographie. L'accent est notamment mis sur la réduction modulaire qui est l'opération élémentaire qui coûte le plus cher dans les implantations cryptographiques en RNS. Un état de l'art des implantations RNS est présenté et analysé, ainsi que les choix d'architecture.

La première contribution de la thèse, présentée au chapitre 2, porte sur l'*inversion modulaire* en RNS. Cette opération difficile en RNS est effectuée généralement grâce à une *exponentiation*, en utilisant le *petit théorème de Fermat*. Cette opération est très longue en RNS, elle représente environ 10% du temps total de calcul d'une multiplication scalaire complète (qui est le calcul principal pour ECC). Le coût en RNS des comparaisons et des divisions nécessaires aux algorithmes de type « Euclide étendu » ont fait qu'il était préférable de garder l'exponentiation de Fermat, notamment dans le cadre d'une implantation matérielle ou aucun matériel supplémentaire n'est requis. Notre proposition est une adaptation RNS de l'*algorithme d'Euclide étendu binaire*, utilisant une certaine astuce pour éviter les comparaisons et pouvant être utilisée sur l'architecture de l'état de l'art des courbes elliptiques en RNS avec quelques petites modifications. Il en résulte un algorithme bien plus rapide que l'état de l'art (de 5 à 12 fois plus rapide sur nos implantations FPGA) et coûtant bien moins d'opérations pour une surface de circuit similaire. Ce travail a fait l'objet d'une première publication à CHES 2013 [19], et sera soumis à un journal dans une version étendue avec les nouveaux résultats présentés dans le chapitre ainsi qu'un nouvel algorithme, qui n'a pas encore été implanté mais qui coûte moins d'opérations que notre première proposition [19].

Le chapitre 3 présente deux contributions. La première a été publiée à ASAP 2014 [20] sur la multiplication modulaire en RNS. La seconde contribution est un nouvel algorithme d'exponentiation, encore non publié. L'idée principale de ce chapitre est de séparer les multi-

plications modulaires RNS en 2 parties, tout d'abord une étape de *décomposition* spécifique à chacun des opérandes, puis une deuxième étape dépendant des deux opérandes pour obtenir le résultat final. Ce qui va permettre de réduire le coût des calculs, c'est que l'on peut factoriser les étapes de décomposition entre plusieurs multiplications modulaires, à chaque *réutilisation* d'une des opérandes. Par exemple, dans l'état de l'art de l'arithmétique RNS, un carré ou une multiplication par une constante coûte aussi cher qu'une multiplication quelconque. Nous proposons un nouvel algorithme de multiplication qui, lui, va être capable de profiter de ces réutilisations. Notre proposition est cependant moins performante que l'algorithme de l'état de l'art sur les multiplications de deux opérandes quelconques, s'il n'y pas de réutilisation. Suivant les séquences de calcul effectuées, nous obtenons des coûts en nombre d'opérations meilleurs que l'algorithme de l'état de l'art sur des grands paramètres cryptographiques. On réduit aussi le nombre de pré-calculs à stocker vis à vis de l'algorithme de l'état de l'art [48]. Les résultats obtenus sont bons pour des exponentiations pour les applications du logarithme discret comme les protocoles de Diffie-Hellman [38] ou Elgamal [44]. Cet algorithme requiert par contre que le nombre premier p, qui définit le corps de base $\mathbb{F}_p$, ait une forme particulière adaptée au RNS. Cette condition empêche son utilisation dans le cadre de RSA. Par contre, en reprenant certaines idées sur la réutilisation, un nouvel algorithme d'exponentiation est proposé, sans condition sur p. Ce dernier algorithme peut, lui, être utilisé avec RSA. Il permet de réduire le nombres d'opérations par rapport à l'exponentiation de l'état de l'art en RNS [48], mais demande plus de pré-calculs.

La troisième contribution de la thèse, présentée au chapitre 4, porte sur un autre algorithme de multiplication modulaire en RNS. Dans cette proposition, nous ne représentons plus directement les valeurs en RNS. En fait nous découpons nos entiers en deux sous-valeurs qui, elles, sont représentées en RNS, en reprenant l'idée de décomposition du chapitre 3. Avec ce découpage, on casse un peu l'aspect non positionnel de la représentation car les deux sous-valeurs satisfont une relation pour retrouver la représentation RNS de notre valeur initiale. Grâce à cette nouvelle représentation et à l'introduction de l'équivalent des premiers de Mersenne pour le RNS, nous obtenons un algorithme qui divise presque par 2 le nombre d'opérations élémentaires par rapport à l'état de l'art, et qui est utilisable pour des implantations de cryptographie sur courbes elliptiques. De même, le nombre de moduli requis est divisé par 2 et le nombre de pré-calculs nécessaires par 4. Un peu à la façon de l'utilisation des premiers pseudo-Mersenne du standard NIST [91] spécialement adaptés à la numération simple de position en base 2, nous proposons une nouvelle façon de choisir le corps de base spécialement adapté au calcul en RNS. Une implantation FPGA de l'algorithme a été faite, presque deux fois plus petite que notre implantation de l'algorithme de l'état de l'art, avec un faible surcoût en temps. Cette contribution fera bientôt l'objet d'une soumission.

Le chapitre 5 est un chapitre à part dans la thèse car il ne concerne pas directement le RNS, bien que traitant de tests de divisibilité et de calcul modulaire. Ce travail a été publié à ComPAS 2013 [18] et est issu d'une collaboration avec un autre doctorant (à l'époque), Thomas Chabrier. Ce travail s'intègre dans un contexte de recodage de clé appelé système de représentation des nombres à base multiple [25, 73] (MBNS pour *multi base number system*) pour accélérer les calculs d'une multiplication scalaire d'une courbe elliptique. Cette accélération intervient au niveau de l'algorithme de multiplication scalaire, et non pas de l'arithmétique : elle est donc parfaitement compatible avec l'utilisation de la représentation RNS. Ce chapitre présente une méthode pour effectuer en matériel des tests de divisibilité de très grands nombres par de petites constantes, en parallèle. L'idée est de pouvoir effec-

tuer à la volée le recodage, ce qui est nécessaire si par exemple on introduit un aléa dans le scalaire pour le protéger contre des attaques par canaux cachés. L'idée principale est de remarquer qu'un certain nombre de factorisations dans les calculs peuvent être effectuées, réduisant le coût de notre opérateur de test de divisibilité. Pour faire apparaître ces factorisations, il ne faut plus travailler seulement en base 2, mais dans une base plus grande de la forme 2^w. Grâce à ces factorisations, les parties qui sont propres à chacun des tests de divisibilité sont très réduites, ne prenant en entrée que $w + \epsilon$ bits ($\epsilon = 3$, 4 ou 5) au lieu de travailler sur les centaines de bits en entrée.

Pour finir, la conclusion propose une synthèse des idées principales qui ont mené aux différentes contributions de cette thèse et présente des perspectives à court terme de travaux dans la lignée de ceux-ci. Le plus long terme est aussi discuté, notamment sur certaines évolutions possibles du RNS et des architectures RNS au vu des résultats obtenus.

Notations

Nous allons ici introduire un certain nombre de notations utilisées dans le reste du document. Tout d'abord, des sigles généraux sont présentés à la table 1 puis les notations utiles à la représentation RNS à la table 2. Les notations présentées pour le RNS ne sont utilisées qu'à partir de la section 1.3 et jusqu'à la fin du chapitre 4. Pour résumer, elles ne sont valables que pour les parties traitant du RNS. Les notations utilisées dans le début du chapitre 1 et dans le chapitre 5 sont détaillées localement car bien moins complexes que celles du RNS.

La complexité des notations du RNS vient du fait que premièrement, il y a en fait une représentation RNS par base (il faut donc pouvoir les différencier) et deuxièmement, on rajoute un étage de détails dans les algorithmes. Par exemple, si nous prenons le cas des courbes elliptiques, un point de la courbe est défini par un ensemble de coordonnées sur un corps fini, par exemple $\mathbb{F}_P$. À ces éléments de $\mathbb{F}_P$ vont être associés des représentations RNS, elles-mêmes constituées de petites composantes entières. De même, les opérations élémentaires comme l'addition sont déclinées en plusieurs niveaux : l'addition de points, d'éléments de F_P, de leur représentation RNS et enfin des composantes d'une représentation RNS. Dans une tentative de hiérarchisation, les notations suivantes sont proposées. Les lettres majuscules en gras représenteront les points d'une courbe, les lettres majuscules fines seront des éléments de $\mathbb{F}_P$, les représentations RNS seront notées comme des vecteurs et enfin leurs composantes seront notées en minuscule (voir table 2).

`ADD`	addition de deux points distincts d'une courbe elliptique
bloc DSP	unité de calcul contenant un multiplieur/accumulateur sur FPGA (DSP pour *digital signal processing*)
BRAM	bloc RAM dans un FPGA
CRT	théorème chinois des restes (*chinese remainder theorem*)
`DBL`	doublement d'un point d'une courbe elliptique
DBNS	système de numération à base double (*double base number system*)
DH	cryptosystème de Diffie-Hellman [38]
DLP	problème du logarithme discret (*discrete logarithm problem*)
DPA	analyse différentielle de la puissance consommée (*differential power analysis*)
DSA	standard du NIST de signature numérique [91] (*digital signature algorithm*)
ECC	cryptographie sur courbes elliptiques (*elliptic curve cryptography*)
ECDH	cryptosystème de Diffie-Hellman sur courbes elliptiques (*elliptic curve DH*)
ECDLP	problème du logarithme discret sur les courbes elliptiques (*elliptic curve DLP*)
ECDSA	standard du NIST de signature numérique ECC (*elliptic curve DSA*)
FF	bascule (*flip-flop*)
FFDLP	problème du logarithme discret sur les corps finis (*finite field DLP*)
FPGA	circuit logique programmable (*field programmable gate array*)
FLT	petit théorème de Fermat (*Fermat's little theorem*)
LSBF	bit de poids faibles en tête (*least significant bits first*)
LUT	table de correspondance (*look-up table*)
`mADD`	addition mixte de deux points distincts d'une courbe elliptique
MBNS	système de numération à base multiple (*multi base number system*)
MRS	représentation en base mixte (*mixed radix system*)
MSBF	bit de poids forts en tête (*most significant bits first*)
NAF	forme non adjacente d'un nombre (*non adjacent form*)
NIST	institut de standardisation américain (*national institue of standards and technology*)
$\mathbf{P}$, $\mathbf{Q}$	points d'une courbe elliptique
pgcd	plus grand commun diviseur
ppcm	plus petit commun multiple
RSA	cryptosystème de Rivest, Shamir et Adleman [102]
RNS	représentation modulaire des nombres (*residue number system*)
SPA	analyse simple de la puissance consommée (*simple power analysis*)
`TPL`	triplement d'un point d'une courbe elliptique
VHDL	langage de description matérielle (*VHSIC hardware description language*)
VHSIC	circuits intégrés très haute vitesse (*very high speed integrated circuits*)

TABLE 1 – Notations générales valables pour tout le document

X, Y	éléments de $\mathbb{F}_P$ ou grands entiers
P	un grand nombre premier définissant le corps de base
ℓ	taille de P, $\ell = \lceil \log_2 P \rceil$ et $\ell \approx 160$–550 bits pour ECC
w	taille d'un mot élémentaire ou d'un canal
n	nombre de mots nécessaires pour représenter une valeur de ℓ bits
k	scalaire de la multiplication scalaire, $k \in \mathbb{N}$ (p. ex. $\mathbf{Q} = [k]\mathbf{P}$)
$\lvert X \rvert_P$	autre notation de X mod P, pour des besoins de concision
$\mathcal{B}_a = (m_{a,1}, \ldots, m_{a,n_a})$	une première base RNS de n_a moduli
$\mathcal{B}_b = (m_{b,1}, \ldots, m_{b,n_b})$	une deuxième base RNS de n_b moduli
$\mathcal{B}_c = (m_{c,1}, \ldots, m_{c,n_c})$	une troisième base RNS de n_c moduli
$m_{a,i}$, $m_{b,i}$, $m_{c,i}$	entiers premiers entre eux 2 à 2, avec $m_{s,i} = 2^w - h_{s,i}$ et $h_{s,i} < 2^{w/2}$ pour $s \in \{a, b, c\}$
$\mathcal{B}_{a\lvert b} = (m_{a,1}, \ldots, m_{b,n_b})$	concaténation des bases $\mathcal{B}_a$ et $\mathcal{B}_b$
$\overrightarrow{(X)_s} = (x_{s,1}, \ldots, x_{s,n_s})$	représente l'élément X en base $\mathcal{B}_s$ (p. ex. $s = a$ ou $s = b\lvert c$) avec $x_{s,i} = \lvert X \rvert_{m_{s,i}}$, peut être abrégé en $\overrightarrow{X_s}$
M_s	produit des moduli de la base $\mathcal{B}_s$: $\prod_{i=1}^{s}$
$M_{s,i}$	le produit de tous les moduli de $\mathcal{B}_s$ sauf $m_{s,i}$: $\frac{M_s}{m_{s,i}}$
$\overrightarrow{T_s}$	$\left(\lvert M_{s,i} \rvert_{m_{s,i}}, \ldots, \lvert M_{s,n} \rvert_{m_{s,n}}\right)$, défini pour le CRT
$\widehat{X}$	modification de la représentation RNS pour les sections 2.2 et 2.3
`EMM`	multiplication élémentaire de w bits modulo un des $m_{s,i}$
`EMA`	addition élémentaire de w bits modulo un des $m_{s,i}$
`EMW`	mot mémoire élémentaire de w bits
`BE`	extension (ou changement) de base
`MR`	réduction de Montgomery RNS de l'état de l'art
`MM`	multiplication de Montgomery RNS de l'état de l'art
`FLT-MI`	inversion modulaire en RNS de l'état de l'art basée sur FLT
`PM-MI`	algorithme d'inversion modulaire plus-minus proposé chap. 2
`BTPM-MI`	variante binaire-ternaire du `PM-MI` proposé chap. 2
`SPRR`	algorithme de multiplication modulaire RNS proposé chap. 3 (*split - partial reduction - reduction*)
`SBMM`	algorithme de multiplication modulaire RNS proposé chap. 4 (*single base modular multiplication*)
`Cox-Rower`	architecture RNS de l'état de l'art pour la cryptographie
`Rower`	élément du `Cox-Rower` calculant modulo l'un des $m_{s,i}$
`Cox`	élément permettant de détecter les dépassements lors des `BE`

TABLE 2 – Notations spécifiques au RNS

Chapitre 1

État de l'art

1.1 La cryptographie asymétrique

La *cryptographie asymétrique* repose sur la notion de *fonction à sens unique à trappe*, c'est à dire une fonction facile à calculer dans un sens, mais dont l'*inverse* est très difficile à calculer sans avoir la clé correspondant à la trappe [38]. Une inversion de la fonction sans clé est censée être impossible, ou du moins à un *coût « raisonnable »* par rapport à l'application visée. On peut par exemple estimer que pour certaines applications, une résistance aux meilleures attaques connues nécessitant plusieurs années sur plusieurs centaines de machines est suffisante. D'un autre côté, pour certaines applications relevant du secret défense, des informations doivent rester confidentielles pour des dizaines d'années, ce qui est aujourd'hui difficile à garantir. Grâce aux fonctions à sens unique à trappe, on peut définir des *protocoles* tels que ceux de Diffie-Hellman [38], qui permettent à deux interlocuteurs d'*échanger une clé commune*. Une autre utilisation est la *signature numérique*, par exemple les standards *Digital Signature Algorithm* (DSA) et *Elliptic Curve Digital Signature Algorithm* (ECDSA) de l'institut américain *National Institute of Standards and Technology* (NIST) [91]. Ce type de cryptographie est aussi appelé *à clé publique* car elle ne nécessite pas de clé secrète partagée au préalable par les deux interlocuteurs. Les travaux de Diffie et Hellman [38], puis de Rivest, Shamir et Adleman [102] font parti des pionniers de la cryptographie asymétrique, et sont toujours très utilisés dans les systèmes de sécurité actuels. On compte aussi parmi ces pionniers Clifford Cocks, qui découvrit RSA [102] avant ses auteurs en 1973 mais dont l'algorithme avait été classifié par le gouvernement britannique puis déclassifié en 1997 [30].

1.1.1 Le cryptosystème RSA

Le cryptosystème RSA, du nom de ses auteurs Rivest, Shamir et Adleman [102], repose sur un problème qui lui est propre, que nous appellerons RSAP, et qui s'énonce ainsi : soient n un entier produit de deux nombres premiers p et q (distincts), et e un nombre premier avec $(p-1)$ et $(q-1)$. Étant donné c, résoudre le RSAP revient à trouver m tel que $m^e \equiv c \bmod n$ ou, autrement dit, cela revient à trouver la racine e-ème d'un élément modulo n.

Ce problème est proche du problème de la factorisation des grands entiers. Notamment, si on est capable de factoriser n en p et q, on peut alors calculer d l'inverse de e modulo $(p-1)(q-1)$, et finalement on calcule $(m^e)^d = m \bmod n$ ce qui résout le RSAP. On ne sait pas par contre si la résolution du RSAP permet de résoudre le problème de la factorisation.

Le protocole de chiffrement RSA est présenté à l'algorithme 1. On considère que le message à chiffrer est un entier naturel inférieur ou égal à $n-1$ (si besoin, découper les messages). Le principal calcul est *l'exponentiation modulaire* et peut être effectué à l'aide d'un des algorithmes présentés en section 1.1.4. À cause des algorithmes de factorisation des grands entiers tels que NFS [70] (*number field sieve*), le nombre d'opérations à effectuer pour casser RSA avec $n = 1024$ bits n'est que de 2^{80}. Ainsi, en 2009, un module RSA de 768 bits a été factorisé [65], ce qui a nécessité des centaines de machines en parallèle et plus de deux ans et demi de calcul. Différents chercheurs étudient sérieusement la possibilité de mettre en œuvre un système pour casser RSA 1024 bits.

Algorithme 1: Protocole de chiffrement RSA [102].

Entrées : Alice diffuse sa clé publique (n, e) et garde secrète d sa clé privée
Entrées : Bob veut chiffrer le message m pour Alice, un nombre dans $[0, n-1]$
Chiffrement
1) Bob récupère la clé publique de Alice, c'est à dire (n, e)
2) Bob calcule l'exponentiation modulaire $c = m^e \bmod n$
3) Bob envoie c à Alice
Déchiffrement
1) Alice calcule $m = c^d \bmod n$

1.1.2 Le problème du logarithme discret, application dans les corps finis

Un autre problème célèbre est celui du *logarithme discret* (DLP). Il peut être vu comme une fonction à sens unique et est utilisé dans certains standards de cryptographie asymétrique. Le problème est défini comme suit (voir [78], pages 103–113) : soient $(G, \times)$ un groupe cyclique fini et $g \in G$ un élément qui le génère. Calculer le logarithme discret de h revient à trouver d tel que $h = g^d$ dans G.

Certains cryptosystèmes sont basés sur le DLP sur des groupes G pour lesquels le logarithme est difficilement calculable. Une idée naturelle de groupe est de choisir les éléments de $\mathbb{F}_p^*$, mais à cause de l'*attaque du calcul d'indices* sur $\mathbb{F}_p^*$ (voir [3]), on l'utilise peu. Si on l'utilisait, l'attaque réduirait par exemple la sécurité d'un système à clé de 1024 bits à seulement 80 bits de sécurité (comme pour RSA). Dans les faits, on choisit plutôt un sous-groupe de $\mathbb{F}_p^*$, noté G, d'ordre premier q. Par exemple, un sous-groupe avec environ 2^{160} éléments ($\log_2 q \approx 160$) pour p de taille 1024 bits (voir par exemple [91]). Comme expliqué à la page 113 de [78], le calcul d'indices ne semble pas pouvoir s'appliquer directement sur ce sous-groupe, on doit alors l'appliquer sur tout $\mathbb{F}_p^*$, le passage de $\mathbb{F}_p^*$ au sous-groupe n'impacte donc pas la sécurité du point de vue de cette attaque. Par contre, il faut faire attention à d'autres attaques sur le sous-groupe, comme celle de Pohlig-Hellman [96] et l'attaque rho de Pollard [97] qui sont en $O(\sqrt{q})$ opérations, donnant un nombre de bits de sécurité correspondant à $(\log_2 q)/2$ lorsque q est premier. Il faut choisir q premier car ces attaques dépendent seulement de la taille du plus gros facteur premier de q. La sécurité conférée par ce problème est donc le minimum des attaques sur $\mathbb{F}_p^*$ et sur le sous-groupe sélectionné. Il est recommandé de choisir des paramètres tels que le système soit aussi résistant aux deux types d'attaque. Par exemple, si $\lfloor \log_2 q \rfloor = 160$ bits, on a donc 80 bits de sécurité pour l'attaque Pohlig-Hellman, on prend alors $\lfloor \log_2 p \rfloor = 1024$ on a aussi 80 bits de sécurité pour le calcul d'indices [91]. Si on compare avec RSA, on se retrouve avec un même niveau de sécurité en traitant avec des éléments de même taille (1024 ou 2048 bits

par exemple), mais le *DLP dans les corps finis* (FFDLP) offre des clés bien plus petites (160 bits contre 1024 par exemple).

Le standard américain DSA [91] du NIST, l'échange de clés de Diffie et Hellman [38] (DH, présenté algorithme 2) et le chiffrement d'Elgamal [44] (algorithme 3) se basent sur le fait que le FFDLP n'est pas résoluble en pratique sur un groupe approprié. Les deux cryptosystèmes sont utilisés dans les conditions décrites dans le paragraphe précédent, c'est à dire q est l'ordre de g modulo p, et q est un grand nombre premier. Le problème utilisé pour ces deux protocoles est en réalité un peu plus faible que le FFDLP car il suffit de savoir forger g^{ab} à partir de g^a et g^b pour casser ces deux protocoles (il est évident que si le FFDLP devient facile, ces cryptosystèmes n'apportent plus aucune sécurité).

Algorithme 2: Échange de clé de Diffie-Hellman [38].

Entrées : Alice et Bob sont d'accord sur (p, q, g) où q et p sont premiers, et q est l'ordre de g dans $\mathbb{F}_p^*$

1) Alice (respectivement Bob) tire un nombre au hasard a (resp. b) dans $[2, q-2]$
2) Alice (respectivement Bob) calcule $u = g^a \bmod p$ (resp. $v = g^b \bmod p$)
3) Alice (respectivement Bob) envoie u (resp. v) à Bob (resp. Alice)
4) Alice (respectivement Bob) calcule la clé partagée $g^{ab} = v^a \bmod p$ (resp. $g^{ab} = u^b \bmod p$)

Algorithme 3: Protocole de chiffrement Elgamal simple [44].

Entrées : Alice garde secrète a sa clé privée et diffuse (p, q, g, g^a) sa clé publique
Entrées : Bob veut chiffrer le message m pour Alice, un nombre dans $[0, p-1]$
Chiffrement
1) Bob récupère la clé publique de Alice, c'est à dire (p, q, g, g^a)
2) Bob choisi k au hasard tel que $1 \leqslant k \leqslant p-2$
3) Bob calcule les exponentiations modulaires $\gamma = |g^k|_p$ et $\delta = |m \cdot (g^a)^k|_p$
4) Bob envoie (γ, δ) à Alice
Déchiffrement
1) Alice calcule $\gamma^{-a} \bmod p$
2) Alice obtient $m = \delta \cdot \gamma^{-a} \bmod p$

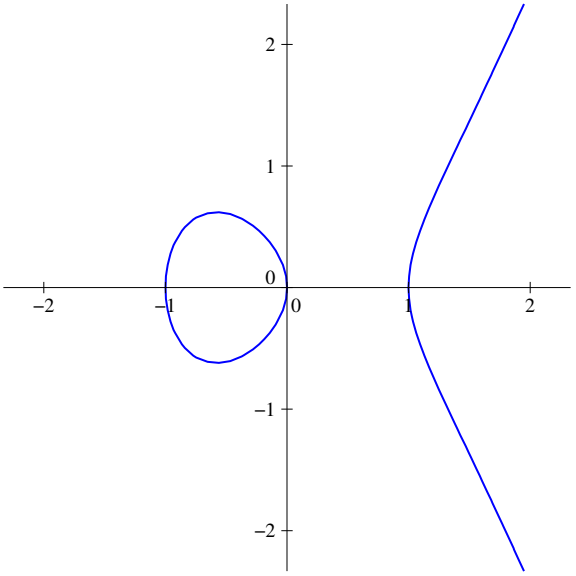

FIGURE 1.1 – Courbe elliptique d'équation $y^2 = x^3 - x$ sur $\mathbb{R}$.

1.1.3 Le problème du logarithme discret sur les courbes elliptiques

Nous allons maintenant nous intéresser à un autre groupe défini à l'aide d'une courbe elliptique et l'utiliser pour le DLP. On va définir ainsi l'ECDLP, *le problème du logarithme discret sur les courbes elliptiques*. L'idée d'utiliser les courbes elliptiques pour la cryptographie a été proposée indépendamment par Koblitz [67] et Miller [80] dans le milieu des années 1980. Cette section rappelle rapidement les définitions et outils nécessaires à la compréhension du cadre de mes travaux, mais son but n'est pas de détailler toutes les mathématiques sous-jacentes. Des ouvrages tels que [57] ou [31] apporteront au lecteur plus de détails sur la cryptographie sur courbes elliptiques (ECC) et les mathématiques sur lesquelles elle repose.

Définition 1. *Une courbe elliptique $E/\mathbb{K}$ est une courbe lisse définie sur un corps $\mathbb{K}$ par l'équation :*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1.1}$$

où $a_1, a_2, a_3, a_4, a_5, a_6 \in \mathbb{K}$.

Le fait que la courbe soit lisse, c.-à-d. sans singularité, se traduit par le fait que $\delta \neq 0$, où δ est le discriminant défini par :

$$\begin{cases} \delta &= -d_2^2\, d_8 - 8\, d_4^3 - 27\, d_6^2 + 9\, d_2\, d_4\, d_6 \\ d_2 &= a_1^2 + 4\, a_2 \\ d_4 &= 2\, a_4 + a_1\, a_2 \\ d_6 &= a_3^2 + 4\, a_6 \\ d_8 &= a_1^2\, a_6 + 4\, a_2\, a_6 - a_1\, a_3\, a_4 + a_2\, a_3^2 - a_4^2 \end{cases} .$$

L'équation 1.1 est communément appelée équation de Weierstrass. Les points de la courbe $E/\mathbb{K}$ sont tout simplement les couples $(x, y) \in \mathbb{K}^2$ qui satisfont l'équation 1.1. La figure 1.1 montre un exemple de courbe elliptique sur $\mathbb{R}$ alors que la figure 1.2 montre 3 exemples sur des corps finis $\mathbb{F}_p$, pour des petits p non utilisables en pratique.

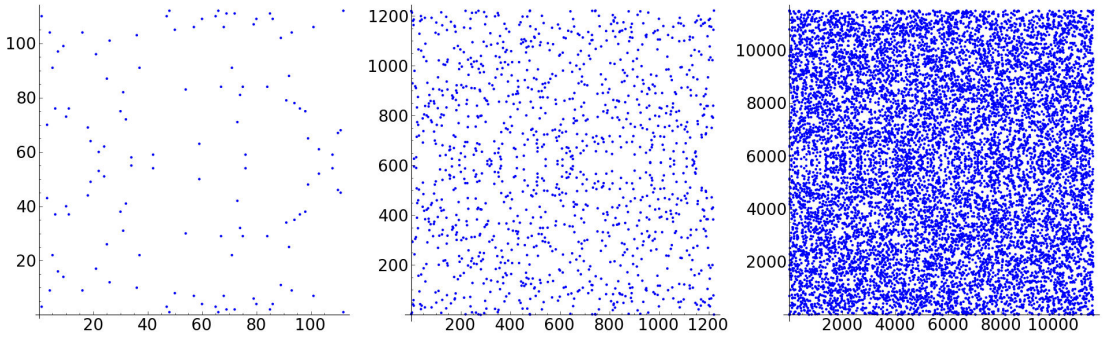

FIGURE 1.2 – Points des courbes définies par $y^2 = x^3 + 3x + 5$ sur $\mathbb{F}_{113}$, $\mathbb{F}_{1223}$ et $\mathbb{F}_{11489}$.

L'équation 1.1 peut être simplifiée sous certaines hypothèses sur le corps, en appliquant un certain changement de variable. Si $\mathbb{K}$ a une caractéristique différente de 2 ou 3 alors le changement de variable :

$$(x, y) \longrightarrow \left(\frac{x - 3a_1^2 - 12a_2}{36}, \frac{y - 3a_1 x}{216} - \frac{a_1^3 + 4a_1 a_2 - 12a_3}{24} \right)$$

permet de réécrire l'équation de $E/\mathbb{K}$ en :

$$y^2 = x^3 + ax + b \tag{1.2}$$

où $(a, b) \in \mathbb{K}^2$. Le discriminant devient alors $\delta = -16(4a^3 + 27b^2)$.

On appelle *forme de Weierstrass courte* l'équation 1.2, dans le cas où la caractéristique est différente de 2 et 3. Nous allons plus précisément travailler sur le corps $\mathbb{K} = \mathbb{F}_p$, où p est un grand nombre premier de plusieurs centaines de bits. Dans le cadre de cette thèse, les corps en petite caractéristique ($\mathbb{F}_{2^m}$) ne seront pas étudiés. En effet, l'arithmétique dans ces corps est très différente du cas $\mathbb{F}_p$ et le système RNS, sous sa forme actuelle, ne semble pas bien adapté à ces corps. Le lecteur curieux d'en savoir plus sur les courbes définies sur $\mathbb{F}_{2^m}$ pourra consulter par exemple [31, 57].

Loi de groupe et ECDLP

Nous allons présenter le groupe sur lequel nous appliquerons le DLP. Une *loi de groupe* peut être définie sur l'*ensemble des points de la courbe*. Cette loi sera notée $+$, $\mathbf{P}+\mathbf{Q}$ est donc l'*addition des points* $\mathbf{P}$ et $\mathbf{Q}$. Ces points sont représentés par leurs coordonnées *affines* $\mathbf{P} = (x_1, y_1)$ et $\mathbf{Q} = (x_2, y_2)$. La loi $+$ applique la règle de la *corde et la tangente* qui, à partir de 2 points $\mathbf{P}$ et $\mathbf{Q}$ de la courbe tels que $\mathbf{P} \neq \pm\mathbf{Q}$, donne un troisième point $\mathbf{R}$ aussi sur la courbe. La corde reliant $\mathbf{P}$ à $\mathbf{Q}$ coupe en fait la courbe en un troisième point (S sur la figure 1.3), le résultat est ensuite obtenu en prenant le symétrique par rapport à l'axe des abscisses (illustration à la figure 1.3 (a)). Le *doublement de point* est le cas spécial de l'addition de point où l'on additionne un point à lui même c'est à dire $[2]\,\mathbf{P} = \mathbf{P}+\mathbf{P}$. Dans ce cas, au lieu de prendre la corde on prend la tangente à la courbe passant par $\mathbf{P}$ (voir figure 1.3 (b)). Une différenciation va donc être faite suivant la nature des opérandes, pour effectuer soit une addition soit un doublement. Pour noter spécifiquement le doublement de la loi de groupe on utilisera `DBL` dans la suite de ce manuscrit et `ADD` pour l'addition de

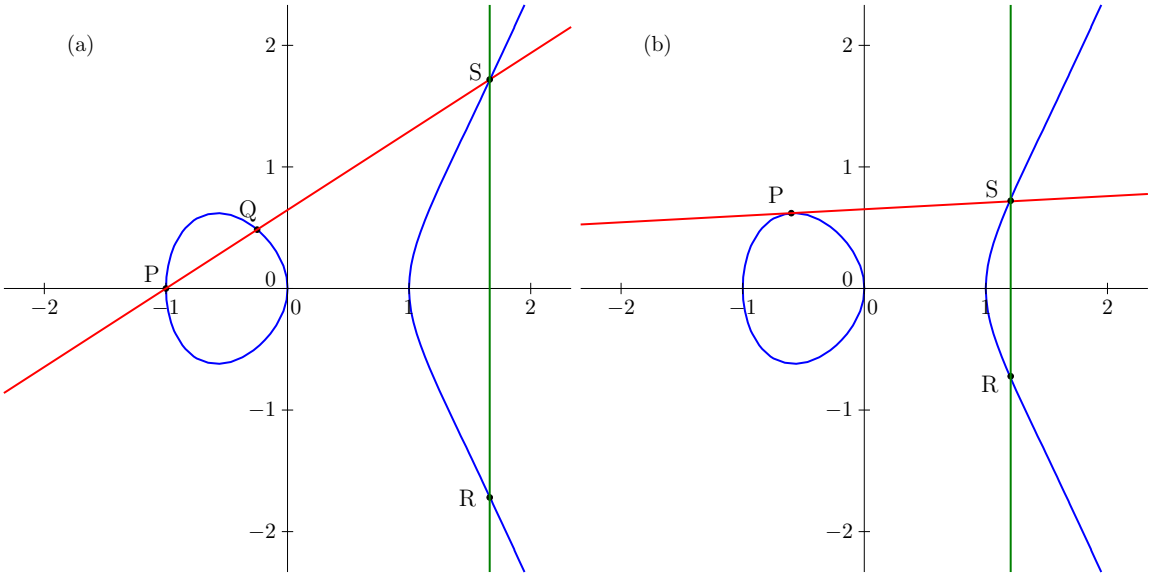

FIGURE 1.3 – Addition (a) et doublement (b) de points sur la courbe $y^2 = x^3 - x$ sur $\mathbb{R}$.

2 points distincts. La description de la loi est faite à la figure 1.3 de façon géométrique sur $\mathbb{R}$, justifiant le nom « règle de la corde et la tangente ». On peut aussi la définir de façon algébrique, et l'adapter à des courbes sur $\mathbb{F}_p$ (illustrées à la figure 1.2). Afin de déterminer complètement la loi de groupe, l'*élément neutre* $\mathcal{O}$ est défini, aussi appelé *point à l'infini*. On ne peut pas lui attribuer de coordonnées affines, c'est ici une sorte de vue de l'esprit pour compléter la loi de groupe. On l'appelle point à l'infini car on peut l'imaginer comme un point qui serait à l'infini en coordonnée y, et pour lequel toutes les droites d'équation $x = c$ (c constant) passerait (voir l'exemple sur $\mathbb{R}$ [31], page 269).

Définition 2 (Loi de groupe en caractéristique $\neq 2$ et 3, pour une courbe d'équation 1.2)**.** *Soient* $\boldsymbol{P} = (x_1, y_1)$, $\boldsymbol{Q} = (x_2, y_2)$ *deux points de* $E/\mathbb{F}_p$. *On définit la loi de groupe sur les points de la courbe comme suit :*

Élément neutre *:* $\boldsymbol{P} + \mathcal{O} = \mathcal{O} + \boldsymbol{P} = \boldsymbol{P}$.

Opposé d'un point *: L'opposé de* $\boldsymbol{P}$ *est* $\boldsymbol{-P} = (x_1, -y_1)$ *(qui est aussi un point de la courbe), on a donc* $\boldsymbol{P} + \boldsymbol{(-\ P)} = \mathcal{O}$.

Addition de points `ADD` *: Si* $\boldsymbol{P} \neq \pm\boldsymbol{Q}$ *alors* $\boldsymbol{P} + \boldsymbol{Q} = (x_3, y_3)$ *est un point de* $E/\mathbb{F}_p$ *où*

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1 \quad \textit{avec } \lambda = \left(\frac{y_2 - y_1}{x_2 - x_1}\right).$$

Doublement de point `DBL` *: Si* $\boldsymbol{P} \neq \mathcal{O}$ *alors* $[2]\boldsymbol{P} = (x_3, y_3)$ *est un point de* $E/\mathbb{F}_p$ *où*

$$x_3 = \lambda^2 - 2x_1, \quad y_3 = \lambda(x_1 - x_3) - y_1 \quad \textit{avec } \lambda = \left(\frac{3x_1^2 + a}{2y_1}\right).$$

Cette loi de groupe va permettre de définir le problème du logarithme discret pour les courbes elliptiques : l'ECDLP. On peut vérifier que la loi présentée est bien interne : si $\mathbf{R} = \mathbf{P} + \mathbf{Q}$, alors $\mathbf{R} \in E/\mathbb{F}_p$. Elle est aussi associative, car on a $(\mathbf{P} + \mathbf{Q}) + \mathbf{R} = \mathbf{P} + (\mathbf{Q} + \mathbf{R})$. L'opération principale des protocoles qui sont basés sur l'ECDLP est la *multiplication sca-*

laire (l'équivalent de l'exponentiation pour le FFDLP). La multiplication scalaire $[k]\mathbf{P}$ revient à accumuler k fois le point $\mathbf{P}$ grâce à l'opération d'addition que nous venons de définir. Des techniques pour effectuer cette opération sont présentées en section 1.1.4. Le problème de l'ECDLP est donc le suivant. Soient $(E/\mathbb{K}, +)$ le groupe des points de la courbe et $\mathbf{P}$ un élément qui génère ce groupe, alors calculer le logarithme discret de $\mathbf{Q}$ revient à trouver k tel que $\mathbf{Q} = [k]\mathbf{P}$.

Si on définit ce groupe là, avec une loi moins naturelle que celle de $(\mathbb{F}_p, \times)$, c'est parce qu'il n'y a pas d'autres attaques connues sur le groupe des points d'une courbe elliptique que les attaques génériques contre le problème du DLP, si certaines précautions sont prises. Ainsi, on ne sait actuellement pas faire d'attaques comme celle du calcul d'indice, alors que celle-ci fonctionne contre le FFDLP. Cependant, les attaques comme celles de Pollard [97] et de Pohlig-Hellman [96] sont toujours applicables, ce qui veut dire que l'ordre du groupe, c'est à dire le nombre de points de la courbe $\#E(\mathbb{F}_p)$, doit toujours être divisible par un grand nombre premier. En pratique, on va choisir une courbe telle que $\#E(\mathbb{F}_p) = lq$ avec l petit, par exemple $l \in \{1, 2, 3, 4\}$ et q un grand nombre premier. On va donc travailler sur le sous-groupe d'ordre q. De plus, le théorème de Hasse [58] (théorème 1) nous garantit que $\#E(\mathbb{F}_p)$ est de la même taille que p. On en conclut donc que pour 80 bits de sécurité, en prenant un p de 160 bits, on va générer des courbes d'environ 2^{160} points. Il suffit de trouver une courbe telle que $\#E(\mathbb{F}_p) = lq$ avec l petit et nous aurons un système offrant *a priori* une sécurité de 80 bits, pour des clés de 160 bits mais aussi des calculs sur des nombres de 160 bits (au lieu de 1024 pour l'application du FFDLP).

Théorème 1 (Hasse [58])**.** *Soit E une courbe elliptique définie sur $\mathbb{F}_p$, alors*

$$p + 1 - 2\sqrt{p} \leqslant \#E(\mathbb{F}_p) \leqslant p + 1 + 2\sqrt{p} \quad .$$

Lors de la sélection de la courbe, certaines précautions seront à prendre en plus sur le cardinal de la courbe. Si par exemple $\#E/\mathbb{K} = p$ (la caractéristique du corps), alors on peut calculer un isomorphisme de la courbe vers $(\mathbb{F}_p, +)$ et calculer le DLP dans ce groupe trivialement [110,114]. De même, pour éviter l'attaque MOV [77], il faut vérifier que l'ordre l du point $\mathbf{P}$ (l'élément générateur du sous-groupe sur lequel on fait la multiplication scalaire) ne divise pas $p^k - 1$, pour k suffisamment grand (pour $l > 2^{160}$, il faut vérifier jusqu'à $k = 20$, voir page 173 de [57]).

L'algorithme 4 présente la version courbes elliptiques du chiffrement Elgamal. On peut de même transposer le protocole de Diffie-Hellman, DSA ou tout autre algorithme basé sur le DLP. Dans la suite du document, lorsque sera évoqué la cryptographie sur courbes elliptiques ou ECC, il sera question des protocoles reposant sur l'ECDLP, utilisant la multiplication scalaire comme opération de base.

Pour conclure, nous allons ici présenter certains éléments de comparaison entre ECC et RSA. Tout d'abord, la table 1.1 (issue de la page 19 de [57]) montre l'évolution prévue des tailles RSA/FFDLP et ECDLP, en se basant sur les temps requis pour les attaques NFS et rho de Pollard. L'écart déjà conséquent pour les niveaux de sécurité conseillés actuellement est amené à se creuser encore et de plus en plus vite (tant que de nouvelles attaques spécifiques aux courbes elliptiques ne seront pas trouvées).

Algorithme 4: Chiffrement avec le cryptosystème Elgamal ECC simple (source [57]).

Entrées : Alice diffuse $(p, E/\mathbb{F}_p, \mathbf{P}, l)$ et sa clé publique $\mathbf{Q}$, d est sa clef secrète
Entrées : Bob veut chiffrer le message $\mathbf{M}$ pour Alice, un point de la courbe
Chiffrement
1) Bob récupère la clé publique de Alice, c'est à dire $\mathbf{Q}$ et les paramètres $(p, E/\mathbb{F}_p, \mathbf{P}, l)$
2) Bob choisi k au hasard tel que $1 \leqslant k \leqslant n-1$
3) Bob calcule les multiplications scalaires $\mathbf{C}_1 = [k]\mathbf{P}$ et $\mathbf{C}_2 = \mathbf{M} + [k]\mathbf{Q}$
4) Bob envoie $(\mathbf{C}_1, \mathbf{C}_2)$ à Alice
Déchiffrement
1) Alice calcule $\mathbf{C}_3 = [d]\mathbf{C}_1$
2) Alice obtient $\mathbf{M} = \mathbf{C}_2 - \mathbf{C}_3$

	Niveau de sécurité par algorithme symétrique (bits)				
	SKIPJACK	Triple-DES	AES	AES	AES
	80	112	128	192	256
Tailles des clés (bits)					
RSA	1024	2048	3072	8192	15360
FFDLP	160	224	256	384	512
ECDLP	160	224	256	384	512
Tailles des éléments (bits)					
RSA	1024	2048	3072	8192	15360
FFDLP	1024	2048	3072	8192	15360
ECDLP	160	224	256	384	512

TABLE 1.1 – Tailles de clés et de l'arithmétique pour les cryptosystèmes asymétriques RSA, FFDLP et ECDLP correspondant aux niveaux de sécurité des cryptosystèmes symétriques SKIPJACK, Triple-DES et AES (source [57]).

Réf.	Implémentation	Slices	DSP	Fréq.	Temps
[56]	$\mathbb{F}_p$ NIST 224 bits	1580	26	487 MHz	365 μs
[118]	RSA-1024	3937	17	400 MHz	1.71 ms

TABLE 1.2 – Comparaison de performances ECC et RSA sur FPGA Virtex 4 XC4VFX12 issue de [56].

Opération	temps moyen	mémoire données	mémoire programme
$[k]\mathbf{P}$ ECC-160	0.81 s	282 octets	3682 octets
$[k]\mathbf{P}$ ECC-224	2.19 s	422 octets	4812 octets
$m^e \bmod n$ RSA-1024	0.43 s	542 octets	1073 octets
$c^d \bmod n$ RSA-1024	10.99 s	930 octets	6292 octets
$m^e \bmod n$ RSA-2048	1.94 s	1332 octets	2854 octets
$c^d \bmod n$ RSA-2048	83.26 s	1853 octets	7736 octets

TABLE 1.3 – Comparaison en temps et en mémoire d'ECC et RSA-CRT sur ATmega128 issue de [55]. L'exposant public e est choisi $e = 2^{16} + 1$.

Implémentation	Client	Serveur
ECC-160 bits	93.7 mJ	93.9 mJ
RSA-1024 bits	397.7 mJ	390.3 mJ

TABLE 1.4 – Consommation d'énergie issue de [123] pour une version simplifié de handshake SSL avec authentification mutuelle utilisant RSA ou ECC sur microcontrôleur basse consommation ATmega128.

Les tables 1.2, 1.3 et 1.4 présentent des exemples de comparaison d'implantations matérielles et logicielles RSA et ECC. La table 1.2 issue de la contribution [56] compare ainsi RSA-1024 et ECC avec le premier du NIST P-224. On rappelle que d'après la table 1.1, ECC défini sur 224 bits fournit le même niveau de sécurité que RSA-2048. Les résultats présentés montrent une forte réduction du nombre de *slices* de Virtex 4 (qui sont les blocs de base du FPGA), contre une utilisation accrue du nombre de DSP blocs (qui sont des accélérateurs arithmétiques). De plus, en terme de vitesse l'implantation ECC est presque 5 fois plus rapide, pour un niveau de sécurité supérieur. La table 1.3 présente des résultats des auteurs de [55], qui ont comparé ECC et RSA sur des implantations sur microcontrôleur ATmega128. On remarque que pour un petit exposant public $e = 2^{16} + 1$, l'exponentiation est plus rapide que la multiplication scalaire. Par contre, l'exponentiation utilisant l'exposant secret d est un ordre de grandeur fois plus lent que la multiplication scalaire. Les mêmes auteurs ont comparé dans [123] la consommation d'énergie d'un *handshake* SSL simplifié utilisant soit RSA-1024 soit ECC-160 toujours sur ATmega128. Ils ont observé une division par 4 de l'énergie consommée pour effectuer le handshake avec ECC-160, tout en gardant une sécurité équivalente.

1.1.4 Techniques classiques de multiplication scalaire et d'exponentiation

Dans cette section sont présentés quelques algorithmes de multiplication scalaire. Ils sont présentés pour une loi de groupe notée additivement (notation pour ECC) mais la transformation en algorithme d'exponentiation est triviale (ainsi par exemple l'algorithme *doublement et addition* 5 peut se réécrire en *carré et multiplication*). Les algorithmes 5 et 6 sont deux variantes de l'algorithme dit *doublement et addition* (ou *double-and-add* en anglais), bits de poids faibles et bits de poids forts en tête, respectivement. Le principe est de parcourir le scalaire k bit à bit afin de calculer le schéma de Hörner correspondant. L'avantage de l'algorithme 5 provient du fait que ses lignes 3 et 4 sont parallélisables, alors que dans l'algorithme 6, ils travaillent sur la même donnée. Par contre l'algorithme 6 ne

Algorithme 5: Multiplication scalaire *doublement et addition* poids faibles en tête (source [57], p. 96).

Entrées : $k = (k_{\ell-1}, \ldots, k_1, k_0)_2$, $\mathbf{P} \in E/\mathbb{F}_p$
Sortie : $[k]\mathbf{P}$
1 $\mathbf{Q} \longleftarrow \mathcal{O}$
2 **pour** i **de** 0 **à** $\ell - 1$ **faire**
3 | **si** $k_i = 1$ **alors** $\mathbf{Q} \longleftarrow \mathbf{Q} + \mathbf{P}$
4 | $\mathbf{P} \longleftarrow [2]\mathbf{P}$
5 **retourner** Q

travaillant que sur une seule valeur, il suffit de stocker un seul point durant le calcul. L'algorithme 7, issu de [125], permet d'accélérer les calculs en réduisant le nombre d'additions, au prix de pré-calculs. Le scalaire k est ici parcouru par fenêtre de w bits (par exemple $w = 4$). Un compromis est alors à faire entre la réduction du nombre d'additions de points et la quantité de points pré-calculés à stocker. En plus de la mémoire pour stocker ces points, le coût du calcul de ces points en début de multiplication scalaire doit être pris en compte si le point de base est régulièrement changé, par exemple pour protéger le circuit contre des attaques statistiques (cf. sous-section 1.1.6). Il existe beaucoup d'autres algorithmes d'exponentiation et de multiplication scalaire (un exemple de plus est donné au chapitre 3). Ces algorithmes proposent par exemple de recoder k dans une certaine représentation pour réduire encore le nombre d'additions de points. Des exemples seront donnés dans la section 1.1.5. Pour plus de détails, le lecteur trouvera par exemple une étude des algorithmes d'exponentiation et de multiplication scalaire rapide dans l'article [51]. Enfin, d'autres algorithmes sont aussi utilisés pour protéger le circuit contre les attaques par canaux cachés, par exemple la multiplication scalaire dite échelle de Montgomery [62] (algorithme 22, chapitre 3).

Algorithme 6: Multiplication scalaire *doublement et addition* poids forts en tête (source [57], p. 97).

Entrées : $k = (k_{\ell-1}, \ldots, k_1, k_0)_2$, $\mathbf{P} \in E/\mathbb{F}_p$
Sortie : $[k]\mathbf{P}$
1 $\mathbf{Q} \longleftarrow \mathcal{O}$
2 **pour** i **de** $\ell - 1$ **à** 0 **faire**
3 | $\mathbf{Q} \longleftarrow [2]\mathbf{Q}$
4 | **si** $k_i = 1$ **alors** $\mathbf{Q} \longleftarrow \mathbf{Q} + \mathbf{P}$
5 **retourner** Q

1.1.5 Réduction du coût de la multiplication scalaire

La figure 1.4 propose une vue sur la *hiérarchie* des opérations d'une multiplication scalaire. On considère généralement deux sous-niveaux : les opérations sur les points `ADD` et `DBL` et les opérations $\pm$, $\times$, $^{-1}$ sur le corps de base $\mathbb{F}_p$ (la plupart des études de l'état de l'art se consacre surtout à la réduction du nombre de multiplications). Afin d'améliorer l'efficacité de la cryptographie sur courbes elliptiques, de nombreuses approches ont été proposées dans la littérature, aux différents niveaux de la figure 1.4. Des améliorations ont aussi été proposées sur la réduction de l'arithmétique des courbes, c'est à dire la réduction

Algorithme 7: Multiplication scalaire par fenêtre fixe de Yao [125].

Entrées : $k = (k_{m-1}, \ldots, k_1, k_0)_{2^w}$ avec $k_i \in [0, 2^w - 1]$, $m = \lceil \frac{\ell}{w} \rceil$ et $\mathbf{P} \in E/\mathbb{F}_p$
Pré-calculs : $\mathsf{T} = (\mathcal{O}, \mathbf{P}, [2]\mathbf{P}, \ldots, [2^w - 1]\mathbf{P})$
Sortie : $[k]\mathbf{P}$

```
1 début
2 |  Q ⟵ T(k_{m-1})
3 |  pour i de m − 2 à 0 faire
4 |  |  pour j de 1 à w faire
5 |  |  |  Q ⟵ [2]Q
6 |  |  Q ⟵ Q + T(k_i)
7 |  retourner Q
```

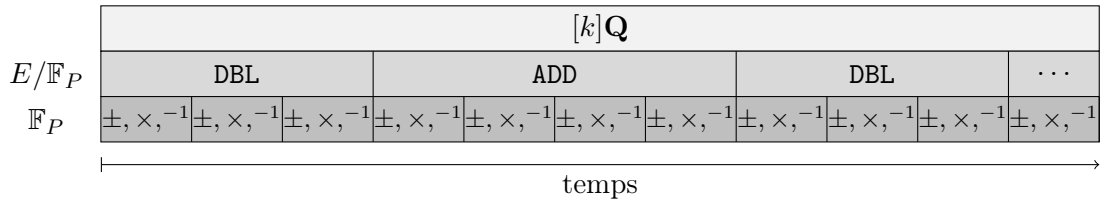

FIGURE 1.4 – Hiérarchie des opérations dans une multiplication scalaire.

du coût des opérations de base telles que l'addition et le doublement de point, ou la proposition de formules pour des opérations plus complexes pour le triplement, par exemple [39], ou le quintuplement avec par exemple [81]. Une façon d'améliorer les formules est de changer de système de coordonnées. En effet, les inversions modulaires étant très coûteuses vis à vis des autres opérations modulaires, l'utilisation de systèmes de coordonnées différents des coordonnées affines a été proposée dans la littérature [28, 32]. Ces systèmes rajoutent un certain nombre de coordonnées, notamment pour éviter de faire une inversion à chaque opération `ADD` ou `DBL`. Par exemple, dans [32] est évoqué un coût d'environ 30 multiplications pour une inversion, ou même 100 multiplications pour les auteurs de [17]. Le retour à une représentation normalisée affine demande une inversion, mais c'est la seule requise sur toute la multiplication scalaire. Par exemple, les coordonnées projectives Jacobiennes font correspondre le point $(x, y) = (X/Z^2, Y/Z^3)$ à (X, Y, Z) avec $Z \neq 0$ pour $(x, y) \neq \mathcal{O}$ (si $Z = 0$ alors $(x, y) = \mathcal{O}$). L'équation 1.2 devient ainsi

$$Y^2 = X^3 + aXZ^4 + bZ^6 \quad .$$

Par exemple, les formules de la table 1.5, issues de [17] permettent de calculer l'addition et le doublement de point sans inversion.

Le site web [17] recense un grand nombre de systèmes de coordonnées, et évalue leur coût en terme d'opérations élémentaires sur le corps en décomptant les opérations nécessaires aux additions, doublements, mais aussi triplements ou encore λ-doublements (formules spéciales lorsqu'on enchaîne les doublements). Une autre façon d'améliorer l'arithmétique de certaines courbes est de les choisir avec une forme particulière permettant des simplifications dans les calculs, autrement dit de changer l'équation de définition d'une courbe pour obtenir de nouvelles formules d'addition et doublement. Par exemple les courbes d'Edwards [43], définies par $x^2 + y^2 = c^2(1 + dx^2y^2)$ avec $cd(1 - c^4d) \neq 0$ mènent à des formules

$\mathbf{P_1} + \mathbf{P_2}$	$2\ \mathbf{P_1}$
$A_1 = Z_1^2$	$A = X_1^2$
$A_2 = Z_2^2$	$B = Y_1^2$
$u_1 = X_1 A_2$	$C = B^2$
$u_2 = X_2 A_1$	$D = Z_1^2$
$S_1 = Y_1 Z_2 A_2$	$S = 2\left((X_1 + B)^2 - A - C\right)$
$S_2 = Y_2 Z_1 A_1$	$M = 3A + aD^2$
$H = u_2 - u_1$	$T = M^2 - 2S$
$I = (2H)^2$	$X_3 = T$
$J = HI$	$Y_3 = M(S - T) - 8C$
$R = 2(S_2 - S_1)$	$Z_3 = (Y_1 + Z_1)^2 - B - D$
$V = u_1 I$	
$X_3 = R^2 - J - 2V$	
$Y_3 = R(V - X_3) - 2S_1 J$	
$Z_3 = \left((Z_1 + Z_2)^2 - A_1 - A_2\right) H$	

TABLE 1.5 – Formules pour courbe elliptique sous forme de Weierstrass courte en coordonnées Jacobiennes (source [17]). Note : $\mathbf{P}_1 = (X_1, Y_1, Z_1)$, $\mathbf{P}_2 = (X_2, Y_2, Z_2)$ et $\mathbf{P}_3 = (X_3, Y_3, Z_3)$.

nécessitant peu d'opérations [16, 17].

On peut aussi accélérer les calculs au niveau de l'algorithme de multiplication scalaire lui même (voir des exemples en section 1.1.4). Des techniques de recodage de la clé ont par exemple été proposées afin de limiter le nombre d'additions de points (souvent plus coûteuses que les doublements). On peut citer par exemple le recodage en forme non adjacente [83] (NAF pour*non-adjacent form*) qui recode la clé avec un nombre limité de 1 (correspondants aux additions) dans la nouvelle représentation. Un autre exemple de recodage est la représentation à base double [39] (DBNS pour *double-base number system*), qui propose de réécrire la clé dans une base double 2 et 3. Ce dernier recodage suppose des formules de triplement de points, qui sont des formules plus efficaces que celles d'un doublement et d'une addition combinées.

Pour finir, les travaux sur l'amélioration de l'arithmétique dans les grands nombres trouvent une application directe dans la cryptographie asymétrique. On notera par exemple les travaux d'implantations de [121] qui testent différentes façons d'implanter l'arithmétique ECC sur GPU, ou encore les travaux [5, 45, 52] qui utilisent la représentation RNS pour accélérer les calculs ECC. Les travaux de cette thèse se placent donc au plus bas niveau, dans l'implantation de l'arithmétique du corps de base. On verra, notamment dans le chapitre 3, que les différents niveaux de la hiérarchie d'une multiplication scalaire (algorithme de multiplication scalaire, opération sur les points et opérations sur le corps) ne peuvent pas être complètement dissociés afin d'être le plus efficace possible, chaque modification sur l'un des niveaux ayant un impact sur les autres.

1.1.6 Sécurité et attaques physiques

Quelques types d'attaques

Les attaques par canaux cachés sont un type d'attaques important, notamment dans un contexte d'implantation matérielle. En effet, ces attaques sont extrêmement dangereuses car elles ne reposent pas sur la qualité du cryptosystème en terme de sécurité purement mathématique, mais sur son implantation. Une implantation matérielle peut être observée ou bien agressée, fautée et ainsi révéler des secrets à l'attaquant, tout en semblant parfaitement sûre mathématiquement. Une implantation cryptographique doit donc être aussi résistante aux deux types d'attaques.

On peut globalement classer en 2 catégories les attaques physiques. D'une part les attaques par observation (ou attaques passives), qui demandent de pouvoir disposer d'un canal de mesure fuitant de l'information, classiquement la consommation de courant [33,69], le rayonnement électromagnétique [4, 100] ou bien le temps d'exécution [68]. D'autre part les attaques par perturbation (ou attaques actives), où l'attaquant va, par exemple, essayer d'introduire des fautes dans le circuit en modifiant un ensemble de bits à l'aide d'un laser.

L'exemple probablement le plus simple d'attaque par observation est l'analyse simple de courant (SPA pour *simple power analysis*), proposée par Kocher *et al.* [69]. Comme présenté dans la section 1.1.4, certains algorithmes d'exponentiation ou de multiplication scalaire effectuent des calculs complètement différents suivant les bits de la clé secrète. Par exemple, l'algorithme 5 effectue dans un cas un simple doublement, et dans l'autre une addition et un doublement. Les additions étant des opérations bien différentes des doublements dans le cas standard (sans contre-mesure spécifique avec des formules unifiées comme dans [23, 61]), les différences de consommation entre les 2 opérations sont très facilement perceptibles. Toujours dans [69] sont proposées des attaques plus sophistiquées, qui à la suite d'un certain nombre de mesures appliquent un traitement statistique afin de retrouver une partie de la clé. On appelle ces attaques DPA, pour *Differential Power Analysis attacks*.

Un exemple connu d'attaque active a été proposé par Boneh *et al.* dans [21]. La proposition est une attaque contre les implantations dites RSA-CRT, où le théorème chinois des restes (détaillé en 1.2.1) est appliqué directement sur la factorisation du module RSA. En injectant une faute dans le calcul effectué sur l'un des facteurs premiers du modulus, on peut retrouver directement la factorisation de celui-ci, et donc casser le système. Une autre exemple est l'attaque *safe error*, présentée dans [128, 129], qui permet d'attaquer certaines implantations d'exponentiation ou de multiplication scalaire. Ces attaques révèlent des informations sur les branchements pris suivant les bits de clé. Cela permet notamment de différencier ces différents branchements, lorsqu'une analyse de courant simple SPA ne suffit pas. Le modèle d'attaquant est par contre ici plus fort que celui d'une SPA car il requiert de pouvoir attaquer avec une très bonne précision le circuit sans le détruire, comme par exemple attaquer un registre contenant une coordonnée d'un point à un moment précis.

Des exemples de contre-mesure

Pour lutter contre les attaques de type SPA, on va généralement chercher à briser le lien entre la consommation de courant et le côté du branchement choisi dans les algorithmes d'exponentiation et de multiplication scalaire. Il existe notamment plusieurs façons d'uniformiser les calculs effectués si le bit de clé k_i vaut 1 ou 0.

La contre-mesure présentée par Joye et Yen dans [62], appelée *échelle de Montgomery* permet d'effectuer une multiplication scalaire en effectuant pour chaque bit de k exactement une addition de points et un doublement. En utilisant cet algorithme (présenté plus en détail dans la section 3.2), on s'attend à ce que la consommation de courant varie très peu suivant que le bit k_i soit à 1 ou 0. De même, la contre-mesure de Möller [86] propose une adaptation de l'algorithme de multiplication par fenêtre de Yao [125] qui effectue toujours w doublements puis une addition, quelque soit la valeur de la fenêtre $(k_i+m-1,\ldots,k_i)$. Cette contre-mesure nécessite donc le pré-calcul de constantes comme pour l'algorithme [125] et requiert en plus un recodage de la clé pour être sûr de ne pas effectuer des opérations irrégulières, comme une addition entre $\mathbf{P}$ et l'élément neutre $\mathcal{O}$.

Une autre contre-mesure visant à uniformiser les calculs est par exemple proposé par Brier et Joye dans [23]. Cette contre-mesure propose l'utilisation de *formules unifiées* pour l'addition et le doublement de point. Autrement dit, on effectue exactement de la même façon l'addition de points et le doublement de point, la consommation ne dépend alors plus des opérations effectuées. On remarque que contrairement aux deux premières contre-mesures [62, 86] qui proposent une protection au niveau algorithme de multiplication scalaire ou d'exponentiation, celle-ci propose une protection directement au niveau des formules pour `ADD` et `DBL`.

Pour se protéger contre les attaques statistiques, on peut aussi citer les contre-mesures de Coron [33] qui sont des contre-mesures classiques pour les implantations ECC. Pour protéger le circuit d'une analyse statistique, l'idée est d'introduire de l'aléa à chaque exécution. Dans [33] est proposé par exemple de rendre aléatoire les coordonnées du point de base $\mathbf{P}$ en coordonnées projectives ou encore d'ajouter des bits d'aléa à la clé sans changer le résultat final.

Une autre façon de se protéger contre les attaques statistiques, mais aussi des attaques de type SPA, est d'utiliser du matériel pour ne plus avoir de lien entre le canal observé et les données ou les calculs effectués. Par exemple, la logique WDDL pour *wave dynamic differential logic* est utilisée par Tiri et Verbauwhede [122] ou encore McEvoy *et al.* [76] pour protéger le système. Dans une implantation WDDL, tout signal s est implanté de paire avec son complémentaire $\overline{s}$. Ainsi, un changement sur s implique un changement sur $\overline{s}$, ce qui rend plus difficile l'analyse de la consommation de courant et le lien qui peut être fait avec les calculs effectués. Nous remarquons que cette contre-mesure est proposée au plus bas niveau en terme d'implantation, elle peut donc être implantée en complément d'autres contre-mesures.

Enfin, pour se protéger contre certaines attaques en faute, Giraud [50] propose de tester le résultat de certains algorithmes de multiplication scalaire ou d'exponentiation avant de retourner un résultat qui donnerait potentiellement des informations à un attaquant. Ce type de contre-mesure doit être utilisé avec un algorithme adapté, comme l'échelle de Montgomery. En effet, certains types d'algorithmes protégés contre la SPA peuvent retourner un résultat juste, malgré qu'une faute ait été commise. Dans ce cas, le test final ne protège plus de la fuite d'information.

Pour conclure sur cette section, ma thèse se plaçant dans un contexte d'implantations matérielles de cryptographie, il est important d'évoquer les attaques par canaux cachés.

Les propositions que nous avons faites dans la thèse portent sur l'accélération des calculs du corps de base, elles sont pour la plupart compatibles avec les contre-mesures qui sont faites aux autres niveaux de la figure 1.4, que ce soit sur l'arithmétique de la courbe ou bien l'algorithme de multiplication scalaire. Par exemple, elles sont compatibles avec les contre-mesures classiques de Coron [33] contre la DPA ou celles de Joye et Yen [62] ou Möller [86] contre la SPA.

1.2 Arithmétique modulaire

Dans cette section, nous allons présenter quelques rappels de base du calcul modulaire, ainsi que des techniques classiques utilisées dans le cadre de la représentation binaire standard de position. Comme nous le verrons dans la suite du document, ces techniques sont souvent à l'origine des techniques utilisées dans le cadre du RNS. Le calcul modulaire spécifique au RNS sera présenté dans la section 1.3, qui lui est consacrée.

1.2.1 Définitions et rappels sur l'arithmétique modulaire

Sans définir dans l'absolu ce qu'est l'*arithmétique modulaire*, on peut la voir comme un type d'arithmétique ne s'intéressant pas directement aux valeurs de nombres sur lesquels on travaille, mais plutôt aux *restes* de leur division par un autre nombre. Le reste de a divisé par b sera écrit $a \bmod b$. L'arithmétique modulaire existe depuis l'antiquité, mais on attribue en général son développement en la version que nous connaissons aujourd'hui à Gauss pour son ouvrage *Disquisitiones Arithmeticae* de 1801. On rappelle maintenant quelques outils nécessaires au calcul modulaire.

Définition 3 (congruences)**.**
Soient $a, b, n \in \mathbb{N}$ avec $n > 0$, alors $a \equiv b \bmod n$ si et seulement si le reste de a divisé par n est égal au reste de b divisé par n. Ils sont alors représentés par la même classe de congruence modulo n. Si $a \equiv b \bmod n$ et $c \equiv d \bmod n$ alors :

- $a + c \equiv b + d \bmod n$;
- $a \cdot c \equiv b \cdot d \bmod n$;
- $a^q \equiv b^q \bmod n$.

On choisit généralement le représentant de la *classe de congruence* comme un élément supérieur ou égal à 0 et inférieur strictement à n (il existe d'autres façons de choisir, par exemple en ayant des représentants négatifs). On rappelle que l'*inverse* de a modulo n, noté a^{-1}, est, lorsqu'il existe, l'unique élément $b \in [0, n-1]$ tel que $ab \equiv 1 \bmod n$. Un tel élément existe si et seulement si a et n sont *premiers entre eux*. Enfin, on rappelle que si m divise n, alors $a \bmod m = (a \bmod n) \bmod m$.

Nous présentons maintenant le *théorème chinois des restes* (qui sera souvent abrégé en CRT). Ce théorème est à la base de la représentation RNS et donc de la grande majorité des travaux de cette thèse. Ce théorème est très ancien (un premier sous-cas a été énoncé par Sun Tsŭ au III^{e} siècle environ, d'après Knuth [66], page 287). L'exemple de Sun Tsŭ peut se traduire comme suit (voir [92]) :

> Soient des objets dont nous ne connaissons pas le nombre.
> Si nous les comptons par 3, il en reste 2.
> Si nous les comptons par 5, il en reste 3.
> Si nous les comptons par 7, il en reste 2.
> Combien y a-t-il d'objets ?

La réponse à ce problème est 23. Cette solution est unique si on considère que le nombre d'objets est inférieur à 105 ($3 \times 5 \times 7$). Nous allons voir comment ce genre de problème se résout dans le cas général, avec l'énoncé du théorème.

Théorème 2 (théorème chinois des restes CRT)**.** *Soient* $\{m_1, \ldots, m_n\}$ *un ensemble d'entiers premiers entre eux deux à deux et* $M = \prod_{i=1}^{n} m_i$*. Soit* $(x_1, \ldots, x_n) \in \mathbb{N}^n$ *alors il existe un unique* $x \in [0, M-1]$ *tel que*

$$\begin{cases} x \equiv x_1 \bmod m_1 \\ x \equiv x_2 \bmod m_2 \\ \vdots \\ x \equiv x_n \bmod m_n \quad . \end{cases} \tag{1.3}$$

Cet élément est défini par

$$x = \left| \sum_{i=1}^{n} \left| x_i \cdot T_i^{-1} \right|_{m_i} \cdot T_i \right|_M \tag{1.4}$$

où $T_i = \frac{M}{m_i}$ *où* $|\cdot|_M$ *représente la réduction modulo* M*.*

Démonstration.
Nous allons d'abord prouver que l'équation 1.4 fourni bien une solution du système 1.3. Les éléments de $\{m_1, \ldots, m_n\}$ étant premiers entre eux 2 à 2, $T_i = \frac{M}{m_i}$ est donc premier avec m_i, son inverse $|T_i^{-1}|_{m_i}$ existe bien. De plus, on remarque que si $i \neq j$, alors T_j est un multiple de m_i, c'est à dire $|T_j|_{m_i} = 0$. Pour résumer,

$$\begin{cases} \left| \left| x_i \cdot T_i^{-1} \right|_{m_i} \cdot T_i \right|_{m_i} = x_i \\ \left| \left| x_i \cdot T_i^{-1} \right|_{m_i} \cdot T_i \right|_{m_j} = 0 \qquad \textit{pour} \quad i \neq j \quad . \end{cases} \tag{1.5}$$

On obtient bien $|x|_{m_i} = x_i$ pour chaque $i \in [1, n]$. Il ne reste plus qu'à prouver l'unicité de la solution. Soient $x, y \in [0, M-1]$, deux solutions du système 1.3. Alors on a $|x-y|_{m_i} = 0$ pour tout $i \in [1, n]$. L'entier $x - y$ est donc multiple de tous les m_i, c'est à dire multiple de leur produit M. Or, puisque $x - y \in [-M+1, M-1]$, le seul multiple de M possible est 0, on en conclut que $x - y = 0$.

□

Ce théorème sera commenté plus en détail dans la section 1.3 consacrée au RNS.

1.2.2 Réduction Modulaire

L'opération caractéristique de l'arithmétique modulaire est le calcul, à partir d'un entier, de sa représentation modulo un certain nombre n, c'est à dire de son reste divisé par n. Cette opération, appelée *réduction modulaire*, peut s'effectuer à l'aide d'une division Euclidienne par exemple, qui retourne quotient et reste. Nous allons voir maintenant quelques techniques utilisées habituellement pour effectuer le calcul du reste sans avoir à calculer le quotient, qui sont efficaces en numération simple de position (les algorithmes sont présentés pour la base 2).

Réduction de Barrett

La réduction de Barrett [15] est une méthode qui va chercher à approcher le quotient de la division de a par p pour obtenir $a \bmod p$. En effet, on rappelle que $a \bmod p = a - \left\lfloor \frac{a}{p} \right\rfloor p$.

La méthode de Barrett propose d'approcher $\left\lfloor \frac{a}{p} \right\rfloor$ par $\left\lfloor \left\lfloor \frac{a}{2^l} \right\rfloor \frac{\mu}{2^l} \right\rfloor$, où $\mu = \left\lfloor \frac{2^{2l}}{p} \right\rfloor$ et l le nombre de bits de p. En faisant une approximation de $\left\lfloor \frac{a}{p} \right\rfloor$, nous sommes sûrs d'obtenir le bon résultat modulo p, à un certain nombre de soustractions par p près (dépendant directement de la précision de l'approximation). Dans la méthode proposée, il suffit de pré-calculer μ, les autres divisions étant en fait juste des décalages. Au final, la réduction calcule :

$$a - \left\lfloor \left\lfloor \frac{a}{2^l} \right\rfloor \frac{\mu}{2^l} \right\rfloor p \quad . \tag{1.6}$$

L'algorithme coûte donc une multiplication de $l \times (l/2)$ bits ($\left\lfloor \frac{\mu}{2^l} \right\rfloor$ ne fait que $l/2$ bits) et une multiplication de l bits, en plus de la soustraction de l'équation 1.6 et de celles nécessaires à corriger le résultat. En utilisant la méthode de Barrett, on obtient un résultat inférieur à $3p$ (voir [15]), on effectue alors finalement une à deux soustractions par p en plus pour obtenir le résultat final.

Réduction de Montgomery

La réduction de Montgomery, initialement proposée en 1985 dans [82], est une méthode de réduction permettant d'éviter l'utilisation de divisions coûteuses, les remplaçant par des opérations très efficaces comme des décalages. Pour pouvoir utiliser ces décalages, la contrainte est d'autoriser la sortie à être un peu supérieure à p pour une réduction modulo p (généralement la sortie est inférieure à $2p$, ce qui implique un bit supplémentaire pour représenter les valeurs). L'algorithme 8 décrit la méthode de Montgomery.

Algorithme 8: Réduction modulaire de Montgomery [82].

Entrées : $a < p^2$, $p < R$ avec $\text{pgcd}(R, p) = 1$
Pré-calcul : $(-p^{-1}) \bmod R$
Sortie : $\omega \equiv a \cdot R^{-1} \bmod p$, $\omega < 2p$
1 $s \longleftarrow a \cdot (-p^{-1}) \bmod R$
2 $t \longleftarrow a + sp$
3 $\omega \longleftarrow t/R$

Cet algorithme prend en entrée un entier $a < p^2$, typiquement le résultat d'un produit de deux éléments inférieurs à p. Pour que cet algorithme soit efficace, il faut que les réductions modulo R et divisions par R soient efficaces. En représentation binaire classique, on prend généralement $R = 2^l$, ainsi une réduction équivaut juste à garder les r derniers bits et les divisions sont de simples décalages. De plus, il est aisé de constater que t défini ligne 2 est un multiple de R en écrivant :

$$t \equiv a + sP \equiv a + a \cdot (-p^{-1}) \cdot p \equiv 0 \bmod R \quad ,$$

de même que de retrouver la borne sur le résultat ω :

$$\omega = \frac{t}{R} = \frac{a + sp}{R} < p \cdot \frac{p + s}{R} < 2p \quad .$$

On peut remarquer que la sortie de l'algorithme n'est pas $a \bmod p$ mais $a \cdot R^{-1} \bmod p$, représentation qu'on appelle parfois domaine de Montgomery. Pour retrouver $a \bmod p$, il suffit de multiplier le résultat de l'algorithme par $R^2 \bmod p$ et réappliquer l'algorithme 8 une fois. Le coût de cette opération peut être négligée, car celle-ci n'intervient qu'une fois tous nos calculs modulaires effectués. De plus, on peut observer que le produit de deux sorties ω_1 et ω_2 de l'algorithme 8 donne un élément $\omega_1 \cdot \omega_2 < 4p^2$ au lieu de p^2. Pour pouvoir maîtriser la taille des entrées et sorties de l'algorithme lorsque les réductions modulaires s'enchaînent, on choisit un élément $R > 2M$, ce qui va permettre de retourner toujours un résultat $\omega < 2p$, même avec une entrée entre p^2 et $4p^2$.

Remarque. La réduction de Montgomery peut être vue comme une première application du théorème chinois des restes. En effet, puisque p et R sont premiers entre eux, si $a < p^2 < pR$ alors d'après le CRT (avec $m_1 = p$ et $m_2 = R$) on a :

$$a = |ap^{-1}|_R \cdot p + |aR^{-1}|_p \cdot R \bmod pR \,,$$

et donc

$$(a - |ap^{-1}|_R \cdot p)/R \equiv |aR^{-1}|_p \bmod pR \,,$$

ce qui nous donne la formule calculée par l'algorithme 8.

Le cas des nombres de Mersenne et pseudo-Mersenne

Les deux algorithmes précédents ne demandent aucune condition sur le modulus p. Dans certains cas, la réduction peut être grandement simplifiée, comme pour les *nombres de Mersenne* qui s'écrivent de la forme $2^\ell - 1$. Par exemple, le standard ECC du NIST [91] propose d'utiliser le corps $F_{P_{521}}$ où $P_{521} = 2^{521} - 1$. Ainsi pour réduire $x = x_1 2^{521} + x_0$ (x_0, $x_1 < 2^\ell$) un nombre de 1042 bits, modulo P_{521}, il suffit de calculer $x \equiv x_1 + x_0 \bmod P_{521}$. On note que $x_1 + x_0$ est un nombre de 522 bits, une soustraction peut être nécessaire pour obtenir le résultat final de la réduction. De même, on réduit en général un nombre de 2ℓ bits en un nombre de ℓ bits.

La même idée peut être généralisée en utilisant ce qu'on appelle les *nombres pseudo-Mersenne*, de la forme $2^\ell - r$, où $0 < r < \sqrt{\ell}$. L'idée de l'utilisation de tels premiers pour ECC a été introduite par Crandall [34]. Si $x = x_1 2^\ell + x_0$ (toujours x_0, $x_1 < 2^\ell$) est un nombre de 2ℓ bits, et $p = 2^\ell - r$ on a maintenant $x \equiv x_1 r + x_0 \bmod p$. Si on pose $s = x_1 r + x_0$, alors on peut remarquer que $\log_2 s = 3\ell/2 + 1$ (en effet $x_1 r$ est de taille $3\ell/2$). En appliquant cette méthode une nouvelle fois, sur s, on obtient à nouveau un nombre de $\ell + 1$ bits auquel on soustrait une ou 2 fois P pour obtenir le bon résultat. L'algorithme 9 résume cette méthode.

Algorithme 9: Réduction modulo un nombre pseudo-Mersenne [34].

Entrées : $p = 2^\ell - r, \quad r < \sqrt{p}, \quad x = x_1 2^\ell + x_0 < p^2, \quad x_0, x_1 < 2^\ell$
Sortie : $\omega \equiv x \bmod p$, $\omega < 2p$

1 $s \longleftarrow x_1 r + x_0 \qquad /^* s = s_1 2^\ell + s_0 {}^*/$
2 $t \longleftarrow s_1 r + s_0$
3 $\omega \longleftarrow |t|_p$

Afin de réduire le coût des multiplications par r dans l'algorithme 9, Solinas [116] remarque que r peut être construit lui aussi de façon particulière, par exemple avec quelques

puissances de 2. Ainsi, dans le standard du NIST est défini $P_{192} = 2^{192} - 2^{64} - 1$, ce qui permet de remplacer les calculs de s et t par 3 additions.

1.2.3 Inversion Modulaire

Dans cette section sont détaillées les techniques classiques pour effectuer l'inversion modulaire, c'est à dire étant donnés p et a, comment calculer l'élément b tel que $ab \equiv 1 \bmod p$. Cette opération est par exemple requise pour ECC, mais on en réduit le nombre au minimum car elle est très coûteuse vis à vis des autres opérations comme la multiplication. Il existe deux grandes familles d'algorithmes pour effectuer cette opération et qui sont présentées ci-dessous.

Inversion avec le petit théorème de Fermat

Le théorème porte le nom de Pierre de Fermat, qui semble-t-il a été le premier à énoncer le théorème qui suit dans une lettre à Frénicle de Bessy (voir [35], page 209) datant du 18 octobre 1640. Le théorème y est énoncé de la façon suivante : « Tout nombre premier mesure infailliblement une des puissances -1 de quelque progression que ce soit, et l'exposant de la dite puissance est sous-multiple du nombre premier donné -1 ». Ce qui peut se traduire par le théorème 3.

Théorème 3 (Petit Théorème de Fermat ou FLT pour *Fermat Little Theorem*)**.**
Soit p un nombre premier et a un nombre qui n'est pas multiple de p. Alors $a^{p-1} - 1 \equiv 0 \bmod p$.

De ce théorème découle directement le fait que si $0 < a < p$ alors $a \times a^{p-2} \equiv 1 \bmod p$, et donc $a^{p-2} \equiv a^{-1} \bmod p$. Ainsi, le calcul de l'inverse modulaire peut se résumer à un grande exponentiation modulaire, implantée par exemple à l'aide de l'un des algorithmes présentés dans la section 1.1.4. Le nombre de tours de la boucle principale est donc en $\log p$. Pour obtenir le coût total, il suffit de multiplier ce nombre par le coût des multiplications internes à la boucle. Les avantages de l'algorithme issu du petit théorème de Fermat (noté FLT) viennent de sa nature d'exponentiation. Ces algorithmes sont très réguliers, et ils demandent principalement de savoir faire des multiplications modulaires et de savoir compter le nombre (fixe) de tours de boucle. Dans un contexte de cryptographie matérielle pour les courbes elliptiques par exemple, les multiplieurs sont déjà implantés pour calculer les opérations nécessaires au doublement de point par exemple. De plus, le contrôle est très simple puisqu'il s'agit juste d'un compteur.

Algorithmes d'Euclide étendus

L'algorithme d'Euclide est à l'origine une méthode permettant de trouver le « plus grand diviseur commun » (pgcd) de deux entiers, proposé par Euclide dans son œuvre *Les Éléments* (livre 7, propositions 1 et 2, voir par exemple l'édition [46]). Cet algorithme permet notamment, dans une version étendue, de trouver les éléments u et v tels que $ua + vb = \text{pgcd}(a, b)$ (identité de Bézout). Supposons que l'algorithme soit calculé pour $0 < a < p$ et p premier, on obtient alors $ua + vp = 1$. Ce qui est équivalent à $ua \equiv 1 \bmod p$, on a donc bien calculé l'inverse de a modulo p. La méthode est détaillée dans l'algorithme 10 (issu de [66]).

L'idée principale de l'algorithme est d'utiliser l'égalité suivante :

$$\text{pgcd}(a, p) = \text{pgcd}(p \bmod a,\ a) \quad .$$

Algorithme 10: Algorithme d'Euclide étendu (source [66]).

Entrées : $a, p \in \mathbb{N}$ avec $\text{pgcd}(a, p) = 1$
Sortie : $U = a^{-1} \bmod p$

$(u_1, u_2, u_3) \longleftarrow (0, 1, p)$
$(v_1, v_2, v_3) \longleftarrow (1, 0, a)$
tant que $v_3 \neq 0$ **faire**
  $q \longleftarrow \left\lfloor \frac{u_3}{v_3} \right\rfloor$
  $(u_1, u_2, u_3) \longleftarrow (u_1, u_2, u_3) - q(v_1, v_2, v_3)$
  $(u_1, u_2, u_3) \longleftrightarrow (v_1, v_2, v_3)$
si $u_1 < 0$ **alors**
  $u_1 \longleftarrow u_1 + p$
retourner u_1

Ainsi en prenant successivement l'opérande le plus grand, modulo le plus petit, on réduit leur taille à chaque tour de boucle, pour finalement arriver au pgcd final (qui vaut 1 dans notre cas). Les opérandes sont d'abord stockés dans les variables u_1, u_2, u_3, v_1, v_2 et v_3. Ces variables sont telles que $u_1 \cdot a + u_2 \cdot p = u_3$ et $v_1 \cdot a + v_2 \cdot p = v_3$ à la fin de chaque tour de boucle. À la fin de l'algorithme, u_3 vaut 1, le résultat est donc bien u_1. On peut remarquer qu'il n'est pas nécessaire de calculer u_2 et v_2, seuls u_1 et v_1 contiendront les valeurs pour arriver à l'inverse, u_3 et v_3 étant requis pour savoir quand arrêter l'algorithme. En moyenne, le nombre de tours de la boucle principale est environ $\frac{12 \log 2}{\pi^2} \ln p \approx 0.58 \log_2 p$, voir par exemple [41,59] si on considère en entrée p et a un élément de $\mathbb{F}_p$. Cela est presque deux fois moins que pour l'algorithme de Fermat, mais il faut en contrepartie être capable de faire des divisions euclidiennes sur des grands nombres de façon efficace.

Une variante célèbre de l'algorithme d'Euclide est sa version dite « binaire », proposée initialement par Stein dans [117]. Cette variante va permettre de s'affranchir du coût des divisions Euclidiennes, pour les remplacer par des divisions par 2 et des tests modulo 2. Ces deux opérations étant triviales en représentation binaire, cet algorithme va permettre une grosse réduction du coût d'une itération de la boucle principale. En contrepartie, plus de tours de boucle principale sont effectués, environ $0.71 \log_2 p$. La version étendue est décrite dans l'algorithme 11 (issu de [66]§ 4.5.2).

Tout au long de cet algorithme, on retrouve l'égalité $u_1 \cdot a \equiv u_3 \bmod p$, et la même chose pour les v_i. La différence vient de la façon dont on fait évoluer ces valeurs intermédiaires. La propriété utilisée pour réduire les opérandes tout en gardant le même pgcd n'est plus $\text{pgcd}(a, p) = \text{pgcd}(p \bmod a,\ a)$, mais on utilise le fait que

$$\begin{cases} \text{pgcd}(u_3, v_3) = \text{pgcd}(u_3/2, v_3) & \text{si } u_3 \text{ pair et } v_3 \text{ impair} \\ \text{pgcd}(u_3, v_3) = 2 \cdot \text{pgcd}(u_3/2, v_3/2) & \text{si } u_3 \text{ et } v_3 \text{ pairs} \\ \text{pgcd}(u_3, v_3) = \text{pgcd}((u_3 - v_3)/2, v_3) & \text{si } u_3 \text{ et } v_3 \text{ impairs.} \end{cases} \tag{1.7}$$

Ainsi, dans la boucle principale on commence par diviser les deux opérandes par 2 tant qu'ils sont pairs. Lorsqu'on divise par 2 la valeur v_3 par exemple, il faut mettre à jour la valeur associée v_1 pour conserver $(v_1/2) \cdot a \equiv (v_3/2) \bmod p$, quitte à rajouter p à v_1 si sa valeur est impaire. Après les deux boucles internes de parité, u_3 et v_3 sont impairs (on est dans le troisième cas de l'équation 1.7) : on fait donc la différence du plus grand et du plus

Algorithme 11: Algorithme d'Euclide étendu binaire de [66]§ 4.5.2.

Entrées : $a, p \in \mathbb{N}, \quad p > 2$ avec $\text{pgcd}(a, p) = 1$
Sortie : $|a^{-1}|_p$

```
1  (u1, u3) ← (0, p),   (v1, v3) ← (1, a)
2  tant que v3 ≠ 1 et u3 ≠ 1 faire
3  |  tant que |v3|2 = 0 faire
4  |  |  v3 ← v3/2
5  |  |  si |v1|2 = 0 alors
6  |  |  |  v1 ← v1/2
7  |  |  sinon
8  |  |  |  v1 ← (v1+p)/2
9  |  tant que |u3|2 = 0 faire
10 |  |  u3 ← u3/2
11 |  |  si |u1|2 = 0 alors
12 |  |  |  u1 ← u1/2
13 |  |  sinon
14 |  |  |  u1 ← (u1+p)/2
15 |  si v3 ≥ u3 alors
16 |  |  v3 ← v3 − u3
17 |  |  v1 ← v1 − u1
18 |  sinon
19 |  |  u3 ← u3 − v3
20 |  |  u1 ← u1 − v1
21 si v3 = 1 alors retourner |v1|p sinon retourner |u1|p
```

petit, puis on applique à nouveau la règle 1.7.

Pour finir est présentée ici une variante de l'algorithme binaire d'Euclide étendu, utilisant une astuce appelée *plus-minus*. Les auteurs de [22] ont proposé de remplacer la comparaison $v_3 > u_3$ par un test modulo 4 dans l'algorithme binaire du calcul du pgcd. Cette astuce est très intéressante pour les représentations dans lesquelles les comparaisons sont difficiles (comme RNS). Il faut surtout être capable de faire de façon efficace des réductions modulo 4. L'algorithme 12, proposé dans [37], est une version étendue de l'algorithme de calcul de pgcd de [22]. L'astuce est de remarquer que si deux entiers sont impairs, alors leur somme ou (exclusif) leur différence est un multiple de 4. Ainsi, au lieu d'effectuer une comparaison entre les valeurs u_3 et v_3 comme dans l'algorithme 11, on teste la valeur modulo 4 de leur somme. Si on obtient 0, on effectue ensuite une division par 4 de cette somme, sinon on divise par 4 la différence. Des petites comparaisons sont effectuées sur les valeurs de contrôle l_u et l_v, mais ces valeurs là sont très petites devant la taille des opérandes (elles sont de taille $\log_2 \log_2 p$). Dans l'algorithme 12, des fonctions `div2` et `div4` sont définies. La fonction `div2` effectue en fait les lignes 5,6,7 et 8 de l'algorithme 11, c'est à dire elle divise par 2 modulo p son entrée. De même `div4` divise par 4 modulo p, sa définition complète dépend de la valeur $|p|_4$. Par exemple, si $|p|_4 = 3$, alors

$$\begin{cases} v_1/4 & \text{si } |v_1|_4 = 0 \\ (v_1 + p)/4 & \text{si } |v_1|_4 = 1 \\ (v_1 + 2p)/4 & \text{si } |v_1|_4 = 2 \\ (v_1 - p)/4 & \text{si } |v_1|_4 = 3\ , \end{cases}$$

ce qui correspond bien à la division par 4 modulo p. Autrement dit, on construit un multiple de 4, en ajoutant un certain nombre de fois p, qui est finalement divisé par 4.

Algorithme 12: Inversion modulaire plus-minus [37].

Entrées : $a, p \in \mathbb{N}$ avec $\text{pgcd}(a,p) = 1, \quad \ell = \lceil \log_2 p \rceil$
Sortie : $|a^{-1}|_p$

$(u_1, u_3) \leftarrow (0, p)$, $l_u \leftarrow \ell$
$(v_1, v_3) \leftarrow (1, a)$, $l_v \leftarrow \ell$
tant que $l_v > 0$ **faire**
  si $|v_3|_4 = 0$ **alors**
    $v_3 \leftarrow v_3/4$
    $v_1 \leftarrow \texttt{div4}(v_1, p)$
    $l_v \leftarrow l_v - 2$
  sinon si $|v_3|_2 = 0$ **alors**
    $v_3 \leftarrow v_3/2$
    $v_1 \leftarrow \texttt{div2}(v_1, p)$
    $l_v \leftarrow l_v - 1$
  sinon
    $v_3^* \leftarrow v_3$, $v_1^* \leftarrow v_1$, $l_u^* \leftarrow l_u$, $l_v^* \leftarrow l_v$
    si $|u_3 + v_3|_4 = 0$ **alors**
      $v_3 \leftarrow (v_3 + u_3)/4$
      $v_1 \leftarrow \texttt{div4}(v_1 + u_1, p)$
    sinon
      $v_3 \leftarrow (v_3 - u_3)/4$
      $v_1 \leftarrow \texttt{div4}(v_1 - u_1, p)$
    si $l_v < l_u$ **alors**
      $u_3 \leftarrow v_3^*$, $u_1 \leftarrow v_1^*$, $l_u \leftarrow l_v^*$, $l_v \leftarrow l_u^* - 1$
    sinon $v \leftarrow v - 1$
si $u_1 < 0$ **alors** $u_1 \leftarrow u_1 + p$
si $u_3 = 1$ **alors retourner** u_1 **sinon retourner** $p - u_1$

Les auteurs de [36] fournissent des résultats d'implantation FPGA Spartan 3 des différents algorithmes d'inversion présentés dans cette section (voir pages 113-116 de [36]). Pour leurs implantations, les algorithmes les plus efficaces sont clairement l'algorithme d'Euclide étendu binaire ainsi que sa variante plus-minus. Les deux implantations sont très proches en termes de temps, et la surface de la version plus-minus se révèle un peu plus petite. L'algorithme classique d'Euclide étendu et celui issu du FLT sont tous deux loin derrière (l'algorithme du FLT restant le plus lent).

1.3 La représentation modulaire des nombres (RNS)

La *représentation modulaire des nombres*, ou RNS (*residue number system*) a été introduite indépendamment par Svodoba et Valach [119] d'un côté et Garner [49] dans les années 50. Proposée initialement pour des problèmes de vérification des calculs, elle a ensuite été utilisée pour des problèmes de traitement du signal [26, 60] puis plus récemment pour différentes implantations de cryptographie asymétrique. Par exemple des travaux sur l'implantation de RSA et FFDLP [10,71,87] (1024–4096 bits), d'autres sur ECC [52,72,104]

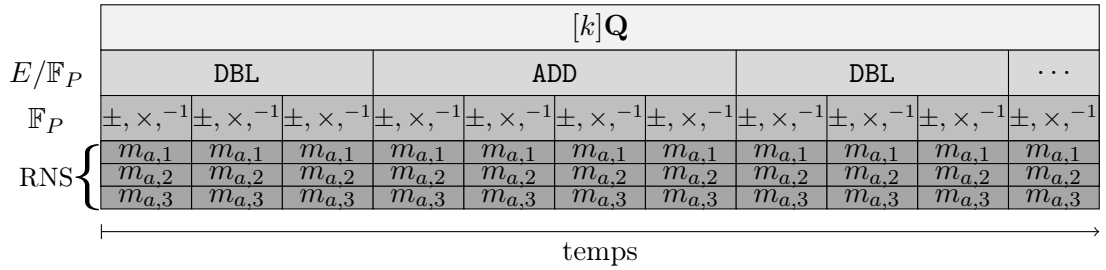

FIGURE 1.5 – Hiérarchie des opérations (fictives) dans une multiplication scalaire en utilisant la représentation RNS.

(160–521 bits) et enfin sur les couplages [27,42] ont été implantés en RNS. Plus de références sont disponibles dans le tableau 1.6. Son intérêt réside dans le fait qu'on peut découper les opérandes et effectuer un certain nombre d'opérations *indépendamment* sur les différents morceaux, c'est-à-dire *sans propagation de retenue*. C'est le cas notamment pour la multiplication, les additions et les soustractions. Ces opérations peuvent donc être effectuées *naturellement en parallèle* et être implantées sur des architectures exploitant cette parallélisation. Cependant, certaines opérations sont difficiles en RNS comme la détection de signe/dépassement, la comparaison, la division Euclidienne ou encore la réduction modulaire. Certaines de ces opérations seront traitées particulièrement dans ce document de thèse. Enfin, une autre difficulté pour le RNS vient du fait que c'est une *représentation non standard*. Les langages de description de matériel (HDL) ainsi que les outils de conception assistés par ordinateur (CAD) ne supportent pas directement la représentation, impliquant un coût de développement important. La figure 1.5 illustre l'intégration du RNS dans la hiérarchie des opérations dans une multiplication scalaire. Dans cette figure, les calculs dans $\mathbb{F}_P$ sont découpés en petits canaux parallèles $m_{a,1}$, $m_{a,2}$ et $m_{a,3}$, afin d'accélérer les calculs. Le lecteur pourra se référer aux ouvrages [92, 115, 120] pour une introduction approfondie sur les différents aspects du RNS.

1.3.1 Définition et premières propriétés

Le CRT, présenté théorème 2, permet de lier directement chacun des éléments de $[0, M-1]$ à un « vecteur » $(x_1, \ldots, x_n)$ (avec $M = \prod_{i=1}^{n} m_i$ et $x_i \in [0, m_i - 1]$ pour tout i). Plus précisément, nous avons une bijection entre les éléments de $\mathbb{Z}/M\mathbb{Z}$ et $(\mathbb{Z}/m_1\mathbb{Z}, \ldots, \mathbb{Z}/m_n\mathbb{Z})$, c'est à dire une nouvelle façon de représenter les éléments modulo M. L'ensemble des m_i permet alors de définir complètement cette représentation et on appelle $(m_1, \ldots, m_n)$ la *base* RNS.

Définition 4. *Soient $\mathcal{B}_a = (m_{a,1}, \ldots, m_{a,n_a})$, où tous les $m_{a,i}$ sont des entiers naturels premiers 2 à 2, et $X \in \mathbb{Z}$. Alors $\mathcal{B}_a$ est une base RNS et on définit la représentation RNS de X en base $\mathcal{B}_a$ par :*

$$\overrightarrow{X} = (x_{a,1}, \ldots, x_{a,n_a}) = (|X|_{m_{a,1}}, \ldots, |X|_{m_{a,n_a}}) \quad . \tag{1.8}$$

De plus, si $X \in [0, M_a - 1]$ où $M_a = \prod_{i=1}^{n_a} m_{a,i}$, et si on définit $T_{a,i} = \frac{M}{m_{a,i}}$ alors

$$X = |X|_{M_a} = \left| \sum_{i=1}^{n_a} \left| x_{a,i} \cdot T_{a,i}^{-1} \right|_{m_{a,i}} \times T_{a,i} \right|_{M_a} \tag{1.9}$$

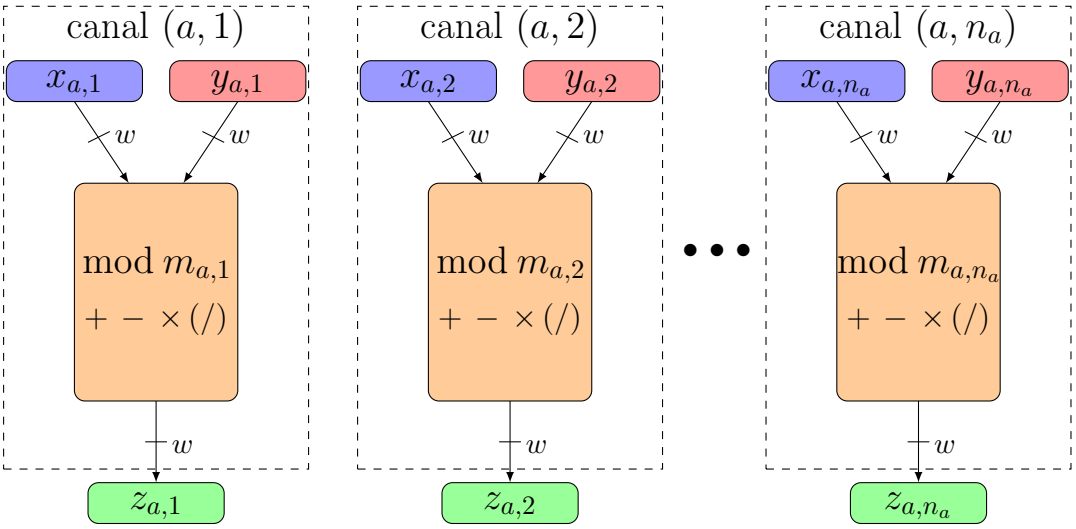

FIGURE 1.6 – Vue globale de la représentation RNS dans la base $\mathcal{B}_a$. La notation $(/)$ désigne la division exacte par un diviseur premier avec chacun des $m_{a,i}$.

permet de retrouver X *à partir de* $\overrightarrow{X}$.

La notation $\cdot$ désigne ici une multiplication entre deux « petites » valeurs de w bits (la taille d'un modulo), alors que $\times$ désigne une multiplication où au moins l'un des deux opérandes est une grande valeur, de l'ordre de $n \times w$ bits ou $(n-1) \times w$ bits. On a introduit dans cette définition une notation plus précise que lors de la définition du CRT en annotant d'un indice a toutes les valeurs correspondant à $\mathcal{B}_a$. Comme on le verra par la suite, plusieurs bases seront définies et ces indices permettront de retrouver rapidement la base dans laquelle nous travaillons. Le détail des notations est présenté dans la section dédiée mais des rappels seront donnés tout au long du manuscrit.

L'arithmétique de base est très simple en RNS, comme indiqué sur la figure 1.6. En fait, en utilisant les propriétés élémentaires du calcul modulaire présentés en section 1.2.1, on voit immédiatement que les additions/soustractions ou multiplications s'effectuent canal par canal avec

$$\overrightarrow{X_a} \diamond \overrightarrow{Y_a} = \left(|x_{a,1} \diamond y_{a,1}|_{m_{a,1}}, \ldots, |x_{a,n_a} \diamond y_{a,n_a}|_{m_{a,n_a}} \right), \tag{1.10}$$

où $\diamond \in \{+, -, \times\}$. Sur la figure 1.6, la division est notée entre parenthèse car cela ne concerne que la division exacte lorsque le diviseur est premier avec M_a. Dans ce cas précis, si on note Z le diviseur, la division revient juste à multiplier par l'inverse de Z modulo $m_{a,i}$ pour tout i, c'est à dire par $\overrightarrow{Z_a^{-1}} = (|Z_a^{-1}|_{m_{a,1}}, \ldots, |Z_a^{-1}|_{m_{a,n_a}})$. On peut remarquer l'équation 1.9 est réduite modulo M_a. Cela signifie que si on calcule une valeur plus grande que M_a, un dépassement se produit et le résultat est réduit modulo M_a. Par exemple, pour éviter un dépassement, deux fois plus de moduli sont nécessaires pour stocker le résultat d'un produit que pour ses opérandes. Par contre la réduction modulo M_a est faite automatiquement lors d'un dépassement, c'est donc une opération très efficace en RNS en base $\mathcal{B}_a$. Nous verrons dans la sous-section 1.3.3 que cette propriété est utilisée pour adapter l'algorithme de Montgomery en RNS.

Les opérations sont faites indépendamment sur chaque canal, sans propagation de retenue (à la différence de la numération simple de position). Ainsi, ces opérations sont parallélisables si nous disposons de plusieurs unités de calcul modulaire. On observe notamment que la multiplication RNS ne coûte que n_a multiplications modulaires élémentaires `EMM`, toutes parallèles. Nous verrons que ce parallélisme est grandement mis à contribution dans les implantations cryptographiques pour son efficacité. De plus, le RNS est une représentation non-positionnelle, il n'y a pas de mot de poids fort ou faible, ils ont tous une place équivalente. Notamment l'ordre des moduli importe peu dans les calculs, on peut ainsi le rendre aléatoire par exemple. Cet aléa a notamment été expérimenté dans l'état de l'art en tant que protection contre certaines attaques par canaux cachés (voir dans la section 1.3.5).

En contrepartie, certaines opérations sont plus difficiles en RNS qu'en numération simple de position. C'est le cas, entre autres, de la comparaison qui est complexe en RNS, car c'est un système non-positionnel. On peut par exemple convertir notre vecteur RNS en une représentation positionnelle comme la représentation MRS (présentée section 1.3.2) ou encore la représentation binaire standard, mais ces conversions sont coûteuses (conversion présentée section 1.3.2 pour le MRS, calcul du CRT pour la conversion vers la représentation binaire standard). Enfin, la réduction modulaire par une valeur différente de M_a est une opération difficile, de même que la division Euclidienne. La réduction étant aussi essentielle que la multiplication dans les calculs pour la cryptographie, nous allons voir dans la suite des techniques mises en œuvre pour réduire son coût, menant à une représentation très efficace dans les implantations cryptographiques. Un outil va d'abord être introduit, appelé *extension de base* (`BE`) qui permet d'effectuer des conversions entre 2 bases RNS.

1.3.2 Extensions de base RNS

Les *extensions de base* ont été introduites par Szabo et Tanaka dans [120]. Une extension de base est une fonction permettant de passer de $\overrightarrow{X_a}$ représenté dans la base $\mathcal{B}_a$ à $\overrightarrow{X_b}$ dans $\mathcal{B}_b$, qui est première avec $\mathcal{B}_a$ (c'est à dire M_a et M_b sont premiers entre eux). Les extensions de base de l'état de l'art permettent d'éviter un retour à la représentation standard. Comme nous allons le voir, il existe deux grandes techniques pour les effectuer, une utilisant encore le CRT, l'autre passant par une représentation intermédiaire appelée *mixed-radix system* (abrégée MRS [49]). On nomme ces conversions « extensions de base » car une fois $\overrightarrow{X_b}$ obtenu, puisque $\mathcal{B}_a$ et $\mathcal{B}_b$ sont premières entre elles, la *concaténation* des 2 vecteurs $\overrightarrow{X_{a|b}}$ est une représentation RNS de X étendue à $n_a + n_b$ restes, dans la base notée $\mathcal{B}_{a|b}$. Dans tous les algorithmes de l'état de l'art, on a toujours $n_a = n_b = n$, les coûts seront parfois donnés dans ce cas précis. Les extensions de base vont être utilisées dans les algorithmes de réduction modulaire de l'état de l'art (cf. section 1.3.3).

Extensions de base RNS via la représentation MRS

La première forme d'extension de base, introduite en [120], propose d'effectuer une conversion de la représentation RNS en base $\mathcal{B}_a$ à la représentation MRS [49] (définie pour la base $\mathcal{B}_a$) puis enfin de convertir cette représentation en base $\mathcal{B}_b$. Nous allons donc tout d'abord définir ce qu'est la représentation MRS en base $\mathcal{B}_a$.

Définition 5. *Soit $(m_{a,1}, \ldots, m_{a,n_a})$ un ensemble de n_a entiers strictement positifs et soient $M_a = \prod_{i=0}^{n_a} m_{a,i}$ et $X < M_a$ (on considère que $m_{a,0} = 1$). Alors la représentation MRS de X dans la base $(m_{a,1}, \ldots, m_{a,n_a})$ est l'unique n-uplet $(x'_{a,1}, \ldots, x'_{a,n_a})$ tel que $0 \leqslant x'_{a,i} < m_{a,i}$ pour tout $i \in [1, n]$ et*

$$X = \sum_{i=1}^{n_a}(x'_{a,i}\prod_{j=0}^{i-1} m_{a,j}) = x'_{a,1} + x'_{a,2}\, m_{a,1} + x'_{a,3}\, m_{a,1}m_{a,2} + \ldots + x'_{a,n_a}\prod_{j=0}^{n_a-1} m_{a,j}\ . \quad (1.11)$$

On peut convertir $(x_{a,1}, \ldots, x_{a,n_a})$ en $(x'_{a,1}, \ldots, x'_{a,n_a})$ de façon efficace, c'est à dire passer du RNS en base $\mathcal{B}_a$ au MRS en base $\mathcal{B}_a$. Tout d'abord, on peut remarquer que $x'_{a,1} = x_{a,1}$ en réduisant l'équation 1.11 modulo $m_{a,1}$. De même, en réduisant l'équation modulo $m_{a,2}$, on obtient $x'_{a,2} = |(x_{a,2} - x'_{a,1}) \cdot m_{a,1}^{-1}|_{m_{a,2}}$. L'algorithme 13 décrit complètement la conversion RNS vers MRS.

Algorithme 13: Conversion RNS vers MRS [120].

Entrée : $\overrightarrow{X} = (x_{a,1}, \ldots, x_{a,n_a})$ en base $\mathcal{B}_a$
Pré-calculs : $|m_{a,i}^{-1}|_{m_{a,j}}$ pour $i < j$
Sortie : $(x'_{a,1}, \ldots, x'_{a,n_a})$

1 $x'_{a,1} \longleftarrow x_1$
2 $x'_{a,2} \longleftarrow |(x_{a,2} - x'_{a,1}) \cdot m_{a,1}^{-1}|_{m_{a,2}}$
3 $x'_{a,3} \longleftarrow \left|\left((x_{a,3} - x'_{a,1}) \cdot m_{a,1}^{-1} - x'_{a,2}\right) \cdot m_{a,2}^{-1}\right|_{m_{a,3}}$
4 $\cdots$
5 $x'_{a,n_a} \longleftarrow \left|\left(\ldots\left(\left(x_{a,n_a} - x'_{a,1}\right) m_{a,1}^{-1}\right) - x'_{a,2}\right) m_{a,2}^{-1}\right) \ldots - x'_{a,n_a-1}\Big) m_{a,n_a-1}^{-1}\Big|_{m_{a,n_a}}$

L'algorithme 13 requiert $\frac{n_a(n_a-1)}{2}$ pré-calculs et autant de multiplications modulaires élémentaires `EMM`. Pour finir l'extension de base, il suffit d'évaluer l'expression de X (équation 1.11) dans chacun des canaux de la seconde base. Autrement dit, on calcule

$$x_{b_k} = \left|x'_{a,1} + x'_{a,2}\, m_{a,1} + x'_{a,3}\, m_{a,1}m_{a,2} + \ldots + x'_{a,n_a}\, T_{a,n_a}\right|_{m_{b_k}} \quad , \quad (1.12)$$

pour tous les $k \in [1, n_b]$. Si on suppose que les $\prod_{j=0}^{i-1} m_{a,j}$ pour tout $i \in [1, n_a]$ sont précalculés, cette dernière étape demande $n_a \times n_b$ `EMM`. Des améliorations sur cette dernière partie ont été proposées par Bajard *et al.* dans [12], en choisissant bien les éléments de $\mathcal{B}_a$ et $\mathcal{B}_b$. En effet, le coût des multiplications modulaires élémentaires `EMM` peut être réduit en prenant des moduli de la forme $2^w - c$ où c est une valeur avec un faible poids de Hamming. De plus, en choisissant bien $\mathcal{B}_a$ et $\mathcal{B}_b$, il est possible de réduire les multiplications de l'équation 1.12 à des combinaisons de décalages et additions.

Bien qu'évoqué ici, ce type d'extension de base ne sera pas utilisé dans nos travaux d'implantation. Tout d'abord le gain obtenu par Bajard *et al.* dans [12] dépend beaucoup de l'architecture. Nous verrons que l'architecture que nous avons choisi comme base de travail, qui est en fait l'architecture de l'état de l'art des implantations ECC en RNS, utilise des multiplieurs embarqués sur FPGA (ceux des blocs DSP). Dans cette architecture, l'addition se révèle même plus chère que la multiplication, comme expliqué dans la section 1.3.5. Remplacer ces multiplications par des additions et des décalages n'est donc pas intéressant dans ce cas. De plus, l'inconvénient de cette méthode est qu'il y a beaucoup de dépendances de données lors de l'algorithme 13, limitant fortement la parallélisation. Les architectures de l'état de l'art étant fortement parallélisées (le parallélisme naturel du RNS est l'un de ses plus gros points forts), cet algorithme n'est pas très adapté. Dans [8], Bajard *et al.* soulignent aussi ce fait, et estiment que la complexité temporelle de leur proposition, en supposant n unités de calcul parallèles, est moins bonne que celle des extensions basées sur le CRT, que nous allons maintenant présenter.

Extensions de base RNS utilisant le CRT

Une autre façon d'évaluer la valeur de X dans la base $\mathcal{B}_b$ à partir $\overrightarrow{X_a}$ est d'évaluer le CRT en $\mathcal{B}_a$ et de le réduire dans la base $\mathcal{B}_b$. On peut d'abord réécrire l'équation 1.9 :

$$X = \sum_{i=1}^{n_a} \left| x_{a,i} \cdot T_{a,i}^{-1} \right|_{m_{a,i}} \times T_{a,i} \; - qM_a \quad , \tag{1.13}$$

où $q = \left\lfloor \left(\sum_{i=1}^{n_a} \left| x_{a,i} \cdot T_{a,i}^{-1} \right|_{m_{a,i}} \times T_{a,i} \right) / M_a \right\rfloor$. L'idée des extensions basées sur le CRT est d'évaluer l'équation 1.13 modulo chacun des éléments de $\mathcal{B}_b$. On remarque d'abord que la somme $\sum_{i=1}^{n_a} \left| x_{a,i} \cdot T_{a,i}^{-1} \right|_{m_{a,i}} \times T_{a,i}$ coûte n_a multiplications pour le calcul de $\left| x_{a,i} \cdot T_{a,i}^{-1} \right|_{m_{a,i}}$ pour tout i, puis n_a multiplications par $T_{a,i}$ pour chaque canal de $\mathcal{B}_b$. Au total, on a donc $n_a n_b + n_a$ `EMM`. Dans les algorithmes utilisés pour la réduction modulaire dans l'état de l'art, on utilise des extensions de base avec $n_a = n_b = n$, ce qui donne un coût de $n^2 + n$ `EMM`. Il est aussi important de noter que si nous avons n unités arithmétiques (toujours avec $n_a = n_b = n$) calculant les multiplications en 1 cycle, alors ce calcul seul demande $n + 1$ cycles. Maintenant, il reste à calculer q, qui est le résultat d'un quotient et qui semble difficile à évaluer. Le matériel de calcul modulaire sur les canaux n'étant pas adapté à ce calcul, nous allons voir comment calculer q en minimisant le coût des opérateurs nécessaires à cette nouvelle opération.

Extensions de base CRT par extra modulo

Une première méthode a été proposée par Shenoy et Kumaresan dans [112]. En fait, ils ont remarqué qu'en ajoutant un modulo supplémentaire m_{a,n_a+1}, il était possible de retrouver q. Posons $\xi_{a,i} = \left| x_{a,i} \left(\frac{M_a}{m_{a,i}} \right)^{-1} \right|_{m_{a,i}} = \left| x_{a,i} \, T_{a,i}^{-1} \right|_{m_{a,i}}$.

D'après l'équation 1.13 on a :

$$q \; M_a = \left(\sum_{i=1}^{n_a} \xi_{a,i} \cdot T_{a,i} \right) - X \; , \tag{1.14}$$

qui pris modulo m_{a,n_a+1} donne

$$|q \; M_a|_{m_{a,n_a+1}} = \left| \left| \sum_{i=1}^{n_a} \xi_{a,i} \cdot T_{a,i} \right|_{m_{a,n_a+1}} - \; x_{m_{a,n_a+1}} \right|_{m_{a,n_a+1}} ,$$

aboutissant finalement à

$$|q|_{m_{a,n_a+1}} = \left| \left| M^{-1} \right|_{m_{a,n_a+1}} \cdot \left| \sum_{i=1}^{n_a} \xi_{a,i} \cdot T_{a,i} \right|_{m_{a,n_a+1}} - \; x_{m_{a,n_a+1}} \right|_{m_{a,n_a+1}} .$$

En choisissant $m_{a,n+1} > q$, on aura alors directement $q = |q|_{m_{a,n_a+1}}$. Par définition, on a $\xi_{a,i} < m_{a,i}$ et donc $\xi_{a,i} \, T_{a,i} < M_a$. Finalement $\sum_{i=1}^{n_a} \xi_{a,i} \, T_{a,i} < n_a M_a$, ce qui prouve que $q < n_a$. Dans les applications cryptographiques, n_a est bien plus petit que la largeur des canaux (typiquement de 2 à 7 bits, contre des canaux de 16 à 64 bits). Le modulo supplémentaire doit juste être supérieur à q, on le choisit donc tel que $m_{a,n_a+1} > n_a$. Cet

algorithme n'est pas très cher, on peut allouer une petite ressource effectuant les calculs sur ce petit modulo supplémentaire qui effectuera les n_a+1 multiplications nécessaires. Les calculs sont seulement sur $\lceil \log_2 n_a \rceil$ bits contre w bits pour les calculs sur les canaux RNS. Il faut noter par contre qu'il faudra effectuer tous les autres calculs cryptographiques aussi sur ce petit canal supplémentaire, il y a donc un surcoût aussi sur les opérations basiques RNS comme la multiplication. Cependant, on verra dans la section 1.3.3 qu'on ne peut pas utiliser cette méthode dans certains cas.

Extensions de base CRT avec approximation

Une autre approche permet de calculer q de l'équation 1.13, ou du moins d'en calculer une bonne approximation. Dans leur travaux [98], Posch et Posch ont remarqué qu'en simplifiant l'expression de q on avait :

$$q = \left\lfloor \sum_{i=1}^{n_a} \frac{\xi_{a,i}}{m_{a,i}} \right\rfloor \quad , \tag{1.15}$$

avec $\xi_{a,i}$ défini comme précédemment. En évaluant une approximation de ce rapport, on aurait donc une approximation du CRT, et donc de $\overrightarrow{X_b}$. Ils ont montré que si on peut limiter X à être $0 < X < (1 - \varepsilon_{max})M_a$, alors on peut toujours s'arranger pour que le calcul approché donne le bon q.

Kawamura *et al.* [64] ont proposé une méthode permettant d'approcher efficacement q en matériel, grâce à un accumulateur sur quelques bits, utilisé seulement lors de l'extension de base (à la différence de la proposition précédente de Shenoy et Kumaresan). Cette approche requiert d'avoir tous les éléments de la base sous la forme $2^w - h_{a,i}$ avec $h_{a,i} < 2^{w/2}$ (c'est-à-dire des pseudo-Mersenne de même taille). Cette condition n'est pas vraiment contraignante car les pseudo-Mersenne sont habituellement choisis pour l'efficacité de la réduction modulaire sur chacun des canaux. De plus, le fait que tous les éléments de la base soient de la même taille est aussi un argument matériel, car il permet d'avoir des unités arithmétiques identiques pour tous les canaux (aux pré-calculs près). Finalement l'approximation se traduit par le calcul de

$$q' = \left\lfloor \sigma_0 + \sum_{i=1}^{n_a} \frac{\texttt{trunc}(\xi_{a,i})}{2^w} \right\rfloor \tag{1.16}$$

où $\texttt{trunc}(\xi_{a,i})$ est une fonction qui approche $\xi_{a,i}$ par ses t bits de poids forts suivis de $w-t$ bits à 0. Si t est petit alors l'évaluation de q revient juste à sommer n_a valeurs de t bits, la division par 2^w sert juste à savoir où se trouve la virgule et borner l'erreur d'approximation (voir les détails dans [64]). Par exemple, les auteurs de [87] ont implanté cette approximation pour des calculs sur RSA 2048 bits, et $t = 8$ était suffisant. Ce calcul était effectué sur un petit accumulateur, en parallèle des calculs sur les unités arithmétiques des canaux.

Avant d'expliquer plus en détail comment l'approximation fonctionne, nous allons présenter l'algorithme 14 d'extension de base, issu de [64]. On peut voir qu'il y a une boucle principale sur la première base (on compte jusqu'à n_a), puis une seconde interne sur la seconde base. Dans cet algorithme, on ne calcule pas d'abord la somme pour trouver q, mais on fait plutôt un calcul à la volée. Ainsi, plutôt que de soustraire q d'un seul coup, il va être soustrait progressivement, remplaçant la multiplication $q(-M_a)$ par une succession

Algorithme 14: Extension de base (`BE`) issue de [64].

Entrées : $\overrightarrow{X_a}$, $\mathcal{B}_a$, $\mathcal{B}_b$, σ_0 (*paramètre fixé du système*)
Pré-calculs : $\overrightarrow{\left(T_a^{-1}\right)_a}, \overrightarrow{(T_a)_b}, \overrightarrow{(-M_a)_b}$
Sortie : $\overrightarrow{X_b}$

1 $\overrightarrow{\xi_a} = \overrightarrow{X_a} \times \overrightarrow{\left(T_a^{-1}\right)_a}, \quad \overrightarrow{X_b} = \overrightarrow{0_b}, \quad \sigma = \sigma_0$
2 **for** $i = 1, \ldots, n_a$ **do**
3 $\quad \sigma = \sigma + \texttt{trunc}(\xi_{a,i})$
4 $\quad q = \lfloor \sigma \rfloor$ /* *Commentaire : q vaut 0 ou 1* */
5 $\quad \sigma = \sigma - q$
6 $\quad$ **for** $j = 1, \ldots, n_b$ **do**
7 $\quad\quad x_{b,j} = |x_{b,j} + \xi_{a,i} \cdot T_{a,i} + q \cdot (-M_a)|_{m_{b,j}}$
8 **retourner** $\overrightarrow{X_b}$

de q soustractions de M_a. Le calcul du CRT sur les éléments de la base $\mathcal{B}_b$ est effectué dans la boucle interne (en matériel sur les unités de calcul modulaire). Pour le coût de l'algorithme, on compte généralement une multiplication RNS pour la ligne 1 (n_a `EMM`), et $n_a \times n_b$ `EMM` pour la ligne 7. Le décompte de la multiplication $q(-M_a)$ dépend des auteurs, on la compte soit comme n_a `EMM` en tant que produit (par exemple dans [48]), soit elle n'est pas comptabilisée comme une multiplication mais comme $q \times n_b$ `EMA`, où `EMA` dénote les additions élémentaires sur chacun des canaux. Les contributions de l'état de l'art sont rares à prendre en compte les additions. Il est difficile d'estimer quelle est la meilleure façon de compter, sachant que $q \times n_b$ dépend du système mais aussi de la valeur qu'on étend à une autre base. On peut cependant borner cette valeur à $n_a \times n_b$ `EMA` pour se situer dans le pire cas. Pour un décompte plus « mathématique », il sera plus simple de compter n_a `EMM` (c'est à dire une multiplication RNS sur la base $\mathcal{B}_a$), alors que dans un cadre d'implantation tel que [52], la valeur $n_a \times n_b$ `EMA` reflétera mieux ce qui est effectué en matériel, le matériel étant conçu pour le pire cas, c'est à dire n_a soustractions sur les n_b canaux.

Les deux théorèmes suivants, extraits de [64], définissent les cadres d'utilisation de l'approximation.

Théorème 4 (issu de [64])**.** *Soit σ_0 tel que $0 \leqslant \varepsilon_{max} \leqslant \sigma_0 < 1$ et supposons X tel que $0 \leqslant X < (1 - \sigma_0)M_a$ alors $q' = q$.*

Théorème 5 (issu de [64])**.** *Soient $\sigma_0 = 0$ et X tel que $0 \leqslant X < M_a$ alors $q' = q$ ou $q' = q - 1$.*

Cela signifie que si X est suffisamment petit par rapport à M_a (le produit des éléments de $\mathcal{B}_a$), on peut faire en sorte que $q' = q$ en choisissant bien σ_0 (qui est directement déterminé par t et les éléments de la base comme nous allons voir). Si ce n'est pas le cas, on obtient alors un résultat approché, mais on sait alors que soit $q' = q$ soit $q' = q - 1$. Le résultat de l'extension de X dans la base $\mathcal{B}_b$ est alors $\overrightarrow{X_b}$ si la conversion a bien fonctionné et $\overrightarrow{(X + M_a)_b}$ s'il y a eu une erreur d'approximation. En fait, on commet 2 erreurs dans l'approximation. La première est due à l'utilisation de `trunc` et dépend donc directement de t. La seconde est due à l'approximation des éléments de la base par 2^w. D'après [64], on a $\tau = \max\left(\frac{\xi_{a,i} - \texttt{trunc}(\xi_{a,i})}{m_{a,i}}\right)$ qui est l'erreur maximale due à `trunc` et

$\delta = \max\left(\frac{2^w - m_{a,i}}{2^r}\right)$ l'erreur maximale due à la division par 2^w. L'erreur maximale globale est alors $\varepsilon_{max} = n_a(\tau + \delta)$. Dans [64], il est prouvé que :

$$\sum_{i=1}^{n_a} \frac{\xi_{a,i}}{m_{a,i}} - n_a(\tau+\delta) \quad < \quad \sum_{i=1}^{n_a} \frac{\texttt{trunc}(\xi_{a,i})}{2^w} \quad < \quad \sum_{i=1}^{n_a} \frac{\xi_{a,i}}{m_{a,i}} \,, \tag{1.17}$$

ce qui permet de déduire le théorème 5. Pour le théorème 4, remarquons d'abord que d'après l'équation 1.14, on a :

$$q + \frac{X}{M_a} = \sum_{i=1}^{n_a} \frac{\xi_{a,i}}{m_{a,i}} \,,$$

et si on rajoute σ_0 tel que $n_a(\tau+\delta) \leqslant \sigma_0 < 1$ de chaque côté dans l'équation 1.17 on obtient :

$$\left(q + \frac{X}{M_a}\right) - n_a(\tau+\delta) + \sigma_0 \; < \; \sum_{i=1}^{n_a} \frac{\texttt{trunc}(\xi_{a,i})}{2^w} + \sigma_0 \; < \; \left(q + \frac{X}{M_a}\right) + \sigma_0 \,. \tag{1.18}$$

En se plaçant dans les conditions du théorème, on a $X < (1-\sigma_0)M_a$, on déduit finalement

$$q \leqslant \sum_{i=1}^{n_a} \frac{\texttt{trunc}(\xi_{a,i})}{2^w} + \sigma_0 \; < q+1$$

qui prouve qu'en prenant la partie entière on obtient bien exactement q.

Nous allons maintenant voir comment ces extensions de base permettent une réduction modulaire efficace en RNS.

1.3.3 Adaptation RNS de l'algorithme de Montgomery

L'algorithme de l'état de l'art de réduction modulaire en RNS est basé sur l'algorithme de Montgomery, présenté à la section 1.2.2. La première adaptation RNS a été proposée dans [99] et optimisée notamment dans [6, 48, 64]. L'algorithme 15 décrit l'algorithme dans sa version initiale, avec une généralisation sur les conditions sur les entrées de l'algorithme grâce à un paramètre α, proposée par Guillermin dans [52].

Algorithme 15: Réduction de Montgomery RNS (`MR`) [99].

Entrées : $(\overrightarrow{X_a}, \overrightarrow{X_b})$ avec $X < \alpha P^2$, $M_a > \alpha P$ et $M_b > 2P$
Pré-calculs : $(\overrightarrow{P_a}, \overrightarrow{P_b})$, $\overrightarrow{(-P^{-1})_a}$, $\overrightarrow{(M_a^{-1})_b}$
Sortie : $\overrightarrow{S} = \overrightarrow{\left|X|M^{-1}|_P\right|_P} + \delta\overrightarrow{P}$ avec $\delta \in \{0,1\}$ in $\mathcal{B}_a$ and $\mathcal{B}_b$

1 $\overrightarrow{Q_a} \leftarrow \overrightarrow{X_a} \times \overrightarrow{(-P^{-1})_a}$ /* *Commentaire : réduction par M_a implicite* */
2 $\overrightarrow{Q_b} \leftarrow \texttt{BE}\left(\overrightarrow{Q_a}, \mathcal{B}_a, \mathcal{B}_b\right)$
3 $\overrightarrow{R_b} \leftarrow \overrightarrow{X_b} + \overrightarrow{Q_b} \times \overrightarrow{P_b}$
4 $\overrightarrow{S_b} \leftarrow \overrightarrow{R_b} \times \overrightarrow{(M_a^{-1})_b}$
5 $\overrightarrow{S_a} \leftarrow \texttt{BE}\left(\overrightarrow{S_b}, \mathcal{B}_b, \mathcal{B}_a\right)$
6 **retourner** $(\overrightarrow{S_a}, \overrightarrow{S_b})$

Les lignes 1, 3 et 4 correspondent aux 3 lignes de l'algorithme 8 où $R = M_a$, plutôt qu'à une puissance de 2. Dans l'algorithme original de Montgomery, on utilise le fait que les réductions modulo 2^w et les divisions par 2^w sont très faciles. Ici, grâce au CRT, calculer dans $\mathcal{B}_a$ opère une réduction implicite du produit $X_a \times (-P^{-1})_a$ par M_a. Par contre, pour pouvoir effectuer une division par M_a, il nous faut passer par une seconde base RNS $\mathcal{B}_b$ (première avec la base $\mathcal{B}_a$), pour laquelle la division exacte par M_a devient triviale comme expliqué au début de la section 1.3. Dans chacune des bases, une seule des 2 opérations est facile, on doit donc avoir recours à des extensions de base pour pouvoir passer de l'une à l'autre.

Dans l'algorithme 8, la sortie de l'algorithme est inférieure à $2P$ et l'entrée pouvait aller jusqu'à $4P^2$. Les travaux présentés dans [52] permettent de généraliser la condition sur l'entrée grâce à un paramètre α. Grâce à cette généralisation, on peut utiliser la technique de réduction paresseuse (ou *lazy reduction*), où on effectue une seule réduction pour calculer par exemple $(AB + CD) \bmod P$, et plus généralement une somme de produits réduite modulo P. Les multiplications et additions étant bien plus efficaces que les réductions en RNS, cette technique est utilisée par exemple en [8] pour proposer des formules ECC adaptées au RNS. En utilisant les mêmes arguments que pour l'algorithme 8, la taille de la sortie est majorée par $2P$ (voir [52] pour la démonstration).

Quelques remarques avant de discuter des améliorations de cet algorithme qui ont été proposées dans la littérature. Tout d'abord, l'algorithme est toujours utilisé dans le cas $n_a = n_b = n$. En effet, son cadre classique d'utilisation est la réduction après un produit de valeurs de $\mathbb{F}_P$ (ou alors d'une somme de produits). L'entrée est de taille de $2\log_2 P$, il nous faut donc déjà $2n$ moduli pour la représenter (impliquant une seconde base $\mathcal{B}_b$). De plus, pour obtenir un résultat de taille $\log_2 P$ (ou plutôt $\log_2 P + \epsilon$), il faut que notre première base soit composée d'au moins n moduli (on a $M_a > \alpha P$). C'est pourquoi on a toujours $n_a = n_b = n$. Ensuite, il faut bien noter que le coût global de l'algorithme est largement dominé par les extensions de base. En effet 2 sont requises, et si on utilise une extension de base de type CRT, on a besoin de $n_a n_b + n_a$ `EMM` sinon via le MRS on obtient $\frac{n_a(n_a-1)}{2} + n_a n_b$ `EMM`. Ces coûts sont donc bien plus élevés que les n_a ou n_b `EMM` requises pour faire les lignes 1, 3 et 4.

Il y a d'abord un choix à faire au niveau des algorithmes d'extension de base. Si nous nous considérons sur une architecture parallèle, nous excluons l'extension de base via MRS pour son manque de parallélisme. Ensuite, nous pouvons remarquer qu'il est impossible d'utiliser l'extension de Posch et Posch [98] pour la première extension de base de manière efficace. Cela vient du fait que l'on ne peut pas calculer $\left||X \times (-P^{-1})|_{M_a}\right|_{m_{a,n+1}}$ efficacement dans le modulo supplémentaire, puisqu'il ne divise pas M_a. On ne peut donc pas évaluer q dans ce cas. Par contre, la seconde extension de base n'inclut pas de réduction implicite, en effet, les calculs lignes 3 et 4 sont faits de telle sorte que le résultat soit borné par $2P$ et comme $M_b > 2P$, aucune réduction n'est opérée. S peut donc bien être calculé dans notre canal supplémentaire, ce qui nous permet d'utiliser l'extension avec extra modulo.

L'extension de base de Kawamura *et al.* [64] peut, elle, être utilisée pour l'extension de base ligne 1 et celle ligne 5. En fait, en ligne 1, on va utiliser le théorème 5 car puisque Q est une valeur prise modulo M_a, on ne peut pas assurer qu'elle soit plus petite que $(1-\sigma_0)M_a$ avec $\sigma_0 > 0$ (on peut avoir $Q = M_a - 1$ par exemple). Le résultat de l'extension de base $\overrightarrow{Q_b}$

est alors soit Q soit $Q + M_a$ dans la seconde base. Il se trouve que cela ne va pas changer le résultat modulo P car on calcule (ligne 3) $\overrightarrow{Q_b} \times \overrightarrow{P_b}$. L'impact de l'erreur d'approximation se retrouve dans la taille de S avec une borne $S < 3P$ au lieu de $S < 2P$ (il suffit de choisir la base $\mathcal{B}_b$ pour un M_b un peu plus grand). Par contre ligne 5, on se retrouve dans le cas d'utilisation du théorème 4 sans erreur d'approximation, car puisque l'on peut borner S suivant P, on peut choisir $\mathcal{B}_b$ et σ_0 pour obtenir le bon résultat. Il est d'ailleurs nécessaire que cette extension de base soit exacte, car elle mène au résultat final (il n'y a pas l'effet de la multiplication par P comme après la première extension de base).

Enfin, une méthode a été proposée par Bajard *et al.* [7] remarquant que, puisque l'erreur de l'approximation de l'extension de base de Kawamura *et al.* effectuée en ligne 1 n'a comme effet que de fournir une sortie $S < 3P$ au lieu de $S < 2P$, alors ils ont proposé de ne même pas calculer q de l'équation 1.13. Ainsi, le résultat obtenu en base $\mathcal{B}_b$ est, au pire, $Q + nM_a$ menant à une sortie $S < nP$. La valeur n étant assez réduite même pour un RSA 2048 (par exemple si $w = 32$ alors $n = 65$) on peut considérer que ce surplus est acceptable. La sortie a donc quelques bits supplémentaires (on devra augmenter la taille de M_a). Par contre, on voit que cette méthode ne s'accorde pas très bien avec la technique de réduction paresseuse car si on calcule par exemple $(AB + CD + EF) \bmod P$ alors la sortie sera de taille $S < 3nP$. Cette technique ne peut-être utilisée que sur la première extension de base, la seconde devant être exacte, elle est souvent combinée à la réduction de Posch et Posch [98] qui elle s'applique dans l'autre extension (voir par exemple [48]).

Le coût de l'algorithme 15 est directement lié au choix des extensions de base. Nous n'allons pas considérer ici le cas d'une extension de base via MRS, pour les raisons déjà évoquées plus tôt. Nous allons utiliser une extension de base via le CRT. On va considérer une configuration avec deux extensions de base `BE` de Kawamura *et al.* [64] (que l'on notera KBE) et une autre avec la combinaison `BE` de Bajard et al. [7]/Posch et `BE` de Posch [98] (notée BPBE). En comptant les multiplications, et en négligeant la multiplication qM_a (considérée comme des additions), on obtient un algorithme avec un coût de $2n^2 + 5n$ `EMM` ($n^2 + n$ par extension de base, plus les lignes 1,3,4). Si on compte la multiplication qM_a, on obtient $2n^2 + 7n$ pour KBE et $2n^2 + 6n$ pour BPBE.

Dans leurs travaux (indépendants), Guillermin [52] et Gandino et al. [48] ont réduit le coût de l'algorithme `MR` en factorisant les calculs dans les extensions de base pour y intégrer directement les lignes 1, 3 et 4. Ainsi, la multiplication ligne 1 de l'algorithme 14 $\overrightarrow{X_a} \times \overrightarrow{\left(T_a^{-1}\right)_a}$ peut se combiner avec la multiplication ligne 1 de l'algorithme 15, $\overrightarrow{X_a} \times \overrightarrow{(-P^{-1})_a}$ en calculant directement $\overrightarrow{X_a} \times \overrightarrow{\left(-P^{-1}T_a^{-1}\right)}_a$. Il faut pour cela pré-calculer $\overrightarrow{\left(-P^{-1}T_a^{-1}\right)}_a$. En fait, on peut faire de même pour la seconde extension de base, en pré-calculant $\overrightarrow{\left(M_a^{-1}T_b^{-1}\right)_b}$. On peut appliquer cette méthode pour KBE et BPBE pour un gain de $2n$ `EMM`. En plus de ces gains, Gandino *et al.* ont prouvé que nous pouvions factoriser encore plus pour la seconde extension de base. En effet, ils proposent de modifier la représentation dans la seconde base en gardant les valeurs multipliées par $\overrightarrow{\left(T_b^{-1}\right)_b}$, ce qui permet ensuite de fusionner les lignes 3, 4 et de directement les intégrer dans la ligne 7 de l'algorithme d'extension de base 14. La modification de la représentation requiert encore quelques pré-calculs. Dans les travaux [48] sont présentés les résultats pour KBE et BPBE. Nous obtenons finalement que le coût de l'algorithme 15 est $2n^2 + 2n$ `EMM` pour KBE et pour BPBE si on compte qM_a comme q additions. Sinon on obtient $2n^2 + 4n$ `EMM` pour

KBE et $2n^2 + 3n$ `EMM` pour BPBE.

1.3.4 Autres algorithmes de réduction

D'autres algorithmes non basés sur celui de Montgomery ont été proposés dans la littérature. L'algorithme de Montgomery semble toujours être la meilleure solution, mais peut-être que des avancées majeures de ces autres formes changeront la donne.

Phillips *et al.* ont proposé dans [95] une façon originale d'effectuer la réduction modulaire, bien qu'elle soit bien moins performante (en théorie et sûrement en pratique) que l'algorithme RNS Montgomery `MR`. Cette méthode, appelée somme des restes, est quand même évoquée car elle propose une vision différente de l'algorithme RNS de l'état de l'art. La méthode ne va pas être présentée de façon aussi détaillée que l'algorithme 15, on présentera ici l'idée utilisée et le coût global de l'algorithme. À notre connaissance, il n'y a pas d'implantation de cet algorithme dans la littérature RNS. Dans cette proposition, on considère la concaténation de $\mathcal{B}_a$ et $\mathcal{B}_b$ comme une seule et unique grande base. X est une valeur issue d'un produit d'éléments de $\mathbb{F}_P$ et l'idée de cette réduction modulaire et d'évaluer directement le CRT modulo P, c'est à dire de calculer :

$$|X|_P \equiv \sum_{i=1}^{2n} \xi_{a|b,i} \times \left|T_{a|b,i}\right|_P + \left|-q\, M_{a|b}\right|_P \quad , \tag{1.19}$$

où $\xi_{a|b,i} = \left|x_{a|b,i} \cdot T^{-1}_{a|b,i}\right|_{m_{a|b,i}}$. On remarque qu'il faut en fait définir notre grande base $\mathcal{B}_{a|b}$ (concaténation de $\mathcal{B}_a$ et $\mathcal{B}_b$), avec plus de bits que pour `MR`. En effet, si on évalue l'équation 1.19, on observe que la somme obtenue est seulement majorée par $P \times \sum_{i=1}^{2n} m_{a|b,i}$ qu'on peut simplifier en $2n2^wP$. De plus, son principal défaut est que l'équation 1.19 nécessite $4n^2$ `EMM`, soit 2 fois plus que l'état de l'art. Enfin, le calcul de q est calculé avec l'approximation de Kawamura *et al.* [64]. L'avantage de cette méthode est que, du fait de sa simplicité, il y a moins de dépendances de données que dans l'algorithme 15.

Enfin, une tentative de transcription de la réduction de Barrett en RNS a été proposée dans [106, 108]. Cette proposition étant récente, je n'ai pas encore pu étudier de façon approfondie la contribution, mais il semble qu'elle soit moins bonne que l'algorithme de Montgomery RNS. Cette méthode impose notamment de pouvoir effectuer des additions et multiplications en représentation standard sur des nombres de la taille de P, ce qui ne semble pas adapté à l'utilisation des architectures habituelles de l'état de l'art RNS. De plus, d'après [106, 108], cet algorithme requiert aussi $4n^2$ `EMM`, soit deux fois plus que l'algorithme de Montgomery RNS.

1.3.5 Implantations RNS

Depuis une quinzaine d'années, le RNS prend de plus en plus d'importance dans les implantations matérielles de cryptographie asymétrique. Dans le tableau 1.6, un bon nombre de ces contributions sont listées, incluant aussi des implantations sur GPU et microprocesseur. Nous allons commenter maintenant quelques unes de ces contributions, et des stratégies vis à vis des paramètres adoptés.

ref.	conf./journ.	aa	usage	implant.	ℓ et $(n \times w)$
[6]	IEEE TC	98	RSA	N	1024 (33×32)
[64]	EuroCrypt	00	RSA	N	1024 (33×32)
[87]	CHES	01	RSA	A 250 nm	672, 1024, 2048, 4096 (22×32), (33×32) (66×32), $(66 \times 32 \star)$
[29]	MWSCAS	03	RSA	F Virtex 2	1024 $(9 \times \{58, \ldots, 64\} \star)$
[10]	IEEE TC	04	RSA	N	1024 (33×32)
[109]	MELECON	06	ECC	Virtex2 Pro	160 (20×30)
[84]	IMA CC	07	RSA	G 7800GTX	1024 $(88 \times 12$ & $44 \times 24)$
[71]	ASSC	07	RSA	P Xtensa	1024 (33×32), 1024 $(17 \times 32 \star)$
[121]	CHES	08	RSA ECC	G 8800GTS	1024 $(16 \times 32 \star)$, 2048 $(32 \times 32 \star)$, 224 (7×32)
[104]	IEEE TCAS I	09	ECC	F Virtex E	160, 192, 224, 256 5 bases/$\mathbb{F}_P$ size e.g. $(30 \times \{23, 28, 30, 35\})$
[72]	TenCon	09	ECC	P Xtensa	192 (7×32)
[52]	CHES	10	ECC	F Stratix I & II	160 (5×34), 192 (6×33), 256 (8×33), 384 (11×35), 521 (15×35)
[27]	CHES	11	Coupl.	F Virtex 6 Stratix 3 Cyclone 2	126, 128, 192 (courbes de Barreto-Naehrig) (8×33), (8×33), (19×33)
[53]	ePrint IACR	11	RSA	Stratix III	1024 $(16 \times 36 \star)$ (33×32)
[105]	ISCAS	11	RSA	N	1024 (33×32)
[127]	Pairing	12	Coupl.	F Virtex 6	126, 128 (courbes de Barreto-Naehrig) (4×67)
[48]	IEEE TC	12	RSA	A 45 nm	1024 (33×32)
[5]	Comp. J.	12	ECC	G 285GTX	224 (15×16)
[9]	Arith	13	RSA	A 250 nm	1024 (33×32), 4096 (65×64)
[94]	DSD	13	RSA	F Spartan 3	1024 $(17 \times 32 \star)$
[45]	IEEE TVLSI	13	ECC	F VirtexE Virtex 2 Pro Stratix II	160, 192, 224, 256 3×56, $(3 \times 66$ & $4 \times 50)$, 4×58, 4×66
[107]	IEEE TCAS	14	RSA	F Virtex6 F Virtex 2	1024 $(66 \times 16$, $33 \times 32)$ 1024 (18×64)
[14]	ePrint IACR	14	ECC	F Kintex 7	256, 521 $(16 \times 17$, $31 \times 17)$

TABLE 1.6 – Implantations de cryptographie asymétrique en RNS. Colonnes : 1 référence, 2 nom conférence/journal, 3 année (19aa/20aa), 4 usage (ECC, Couplages, RSA), 5 implantation (N pour des résultats Non reportés, FPGA, ASIC CMOS, GPU, Processeur), 6 paramètres généraux (taille de $\mathbb{F}_P$ et caractéristiques des bases RNS). $\star$ note une implantation RSA-CRT.

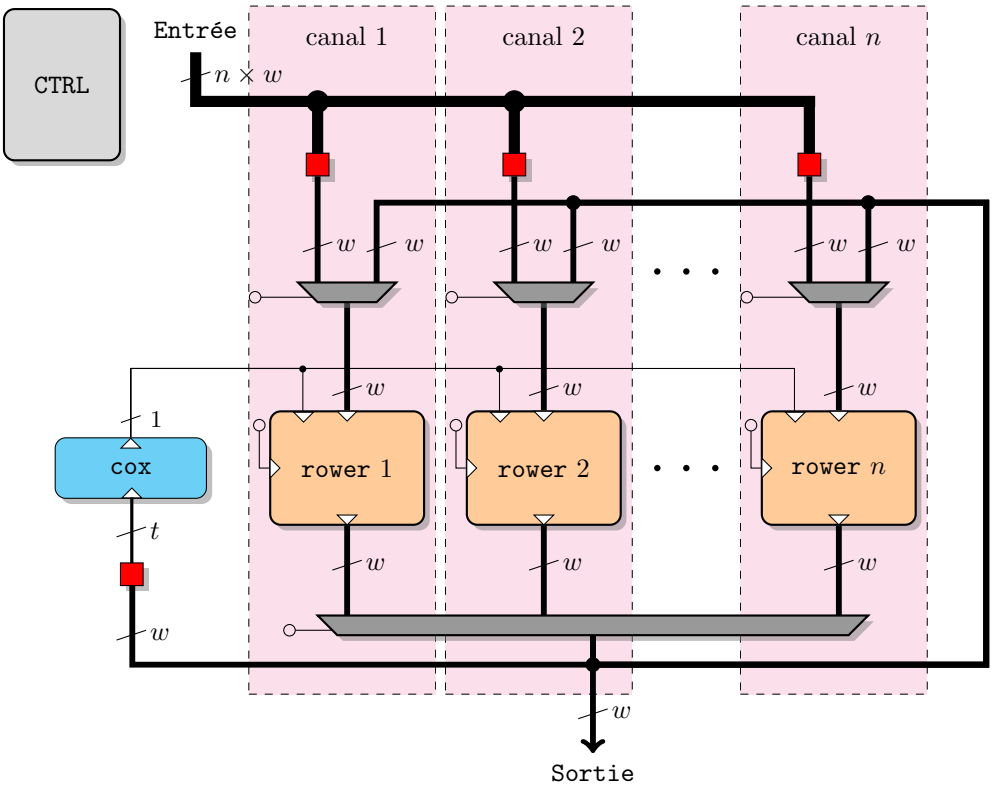

FIGURE 1.7 – Architecture `Cox-Rower` pour le calcul ECC de [52] (adaptation de [64]).

La première implantation matérielle complète de RSA en RNS se trouve dans la contribution [87], qui est une implantation de l'architecture dite `Cox-Rower` permettant le calcul modulaire efficace en RNS. Ce travail fait suite à la proposition d'architecture [64], ou était proposée l'astuce de Kawamura *et al.* et présenté le `Cox-Rower`. La figure 1.7 représente une version de l'architecture de Kawamura *et al.* utilisée dans le cas de calculs ECC dans [52].

Cette architecture `Cox-Rower` est composée d'unités chargées des calculs sur les différents canaux des bases $\mathcal{B}_a$ et $\mathcal{B}_b$, appelés `Rower`s dans la figure 1.7, et d'un petit accumulateur `Cox` calculant q pour l'extension de base (présenté figure 1.8). Dans cette thèse, nous conserverons les termes anglais `Cox` et `Rower` pour garder la cohérence avec la notation de la littérature sur le sujet et ainsi faciliter la compréhension. On pourrait traduire ces termes par *barreur* pour le `Cox` et rameurs pour les `Rower`s. Comme me l'a expliqué le professeur Kawamura rencontré lors de CHES 2013, le nom vient du fait que le `Cox` rythme le travail des rameurs ou `Rower` durant l'extension de base (le signal passe à 1 pour ordonner la soustraction de M_a dans l'algorithme 14, ligne 7). Le `Cox` laisse le véritable effort de calcul aux `Rower`s, il ne doit donc pas être trop grand vis à vis de ceux-ci, mais il est indispensable pour arriver à destination (c.-à-d. avoir une approximation correcte).

Dans l'architecture figure 1.7, les petits carrés indiquent de simples sélections de bits (par exemple pour le `Cox`, les t bits de poids forts) et les petits cercles (—○) sont des signaux de contrôle. Cette figure illustre le principe général des principales architectures de l'état de l'art, qui sont souvent avec n `Rower`s pour n canaux (c'est à dire autant de `Rower`s que d'éléments dans une base). Le `Cox` est illustré figure 1.8. Comme déjà expliqué précédemment, c'est une petite unité composée d'un accumulateur de t bits (t choisi afin

de calculer de façon exacte q).

Il y a eu différentes variations d'architectures `Cox-Rower`. Par exemple, dans la version originale de Kawamura *et al.* [64], une propagation de retenue entre les différents blocs est instanciée pour effectuer la conversion RNS/représentation binaire standard. Guillermin, dans ses travaux [52], montre qu'il n'est pas nécessaire d'avoir ces retenues entre les blocs en effectuant une conversion un peu plus lente. Ceci dit, les temps de conversion sont négligeables devant les temps de calcul de la multiplication scalaire ou d'une exponentiation.

On peut trouver un autre exemple dans le papier de Nozaki *et al.* [87] où les auteurs ont choisi d'implanter un `Cox` par `Rower` (en fait un `Cox` est inclus dans chacun des `Rower`s), pour limiter la sortance (*fanout* en anglais) du `Cox` ainsi que le routage. De plus, pour cette implantation de RSA, 11 `Rower`s sont implantés permettant d'effectuer les calculs sur 22, 33 et 66 moduli. Cette contribution met donc aussi en valeur l'aspect modularité du RNS (leur architecture permet d'effectuer des calculs RSA de 672 à 2048 bits, et jusqu'à 4096 si on utilise une implantation de type RSA-CRT). Enfin, afin d'effectuer l'extension de base de Kawamura *et al.*, les auteurs ont opté pour une structure en anneau pour les `Rower`s, c'est à dire que chaque `Rower` a sa sortie reliée à un `Rower` voisin.

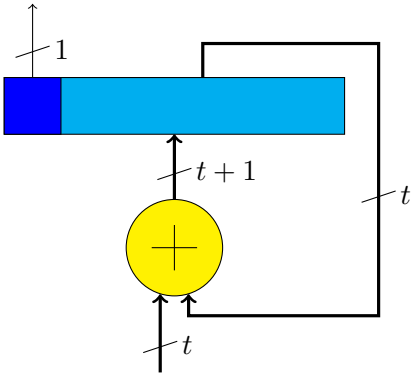

FIGURE 1.8 – `Cox` présenté dans [64].

Du côté des contributions s'intéressant spécifiquement aux implantations ECC en RNS, la première implantation a été proposée en [109] en 2006. Celle-ci s'éloigne assez de ce qu'avaient proposé Nozaki *et al.* [87] et Kawamura *et al.* [64] car ils n'utilisent pas de réduction modulaire `MM` classique, mais utilisent un passage par la représentation standard binaire. Cette stratégie ne semble pas être efficace, car les performances obtenues sont bien inférieures aux propositions basées sur l'utilisation de l'algorithme de Montgomery RNS (comme celle [52]). Les mêmes auteurs ont d'ailleurs choisi d'utiliser les méthodes classiques d'extension de base pour leurs autres contributions par la suite [45, 105, 107].

La contribution [52] propose une des implantations ECC les plus rapides, et protégée contre certaines attaques (par exemple contre les attaques SPA grâce à l'utilisation de l'échelle de Montgomery [62]). Cette implantation ne nécessite pas de forme particulière pour P, à la différence d'une implantation du standard du NIST [91] par exemple. Les implantations faites dans cette thèse sont très similaires à cette implantation. En effet, dans le cadre des implantations RNS, cette contribution est la plus rapide en étant protégée. Elle obtient d'ailleurs des résultats assez proches de [45], qui est l'implantation RNS la plus

rapide à notre connaissance. Plus de détails seront donnés à propos de cette contribution qui a servi de base architecturale pour nos implantations.

La contribution [45] propose une implantation encore plus rapide, toujours en RNS, mais ne peut être comparée équitablement avec la contribution [52] car elle ne propose aucune contre-mesure contre les attaques par canaux cachés. Ainsi, l'algorithme de multiplication scalaire utilisé est l'algorithme 6 *doublement et addition*. De plus, les formules d'addition et de doublement de points ne sont pas les mêmes que [52]. Il est donc impossible, ou très difficile, de comparer les choix portant uniquement sur l'arithmétique RNS.

Choix des paramètres

Les propositions de Guillermin [52] et Esmaeildoust *et al.* [45] dénotent des différences importantes dans les choix d'implantation et de la meilleure utilisation du RNS. Nous allons discuter ici ces choix, en commençant par comparer l'approche des ces 2 papiers.

Guillermin a choisi d'utiliser des tailles de mots (c'est à dire de canaux) $w \leqslant 36$ bits sur les FPGA Stratix pour utiliser au mieux leurs multiplieurs embarqués dans les blocs DSP. L'idée est donc d'avoir des canaux très rapides, une architecture très parallélisée avec des opérations sur des petits mots. De l'autre côté, Esmaeildoust *et al.* [45] choisissent de gros moduli de 50 à 66 bits sur la même architecture, limitant donc la parallélisation mais permettant de réduire le coût des réductions modulaires (on rappelle que chaque `MM` coûte $2n^2 + O(n)$ `EMM`). De plus, elle permet de choisir des canaux pour lesquels les opérations sont très efficaces, et permet aussi d'utiliser des bases très bien choisies, à la façon des travaux [12]. Par contre, une telle stratégie mène à des fréquences bien plus basses que celle de Guillermin (un facteur 3 au niveau des fréquences en faveur de Guillermin). Des résultats de comparaison sont disponibles dans [45], mais ne déterminent pas de vrai vainqueur. En effet, de prime abord les résultats fournis semblent un peu plus à l'avantage de Esmaeildoust *et al.*. Mais c'est sans compter que, premièrement, les travaux de [45] n'implantent que l'algorithme doublement et addition présenté algorithme 5, donc moralement un quart d'opérations en moins, et deuxièmement les temps ne sont comparés que pour les plus petites courbes, leur stratégie semble très peu adaptée aux hauts niveaux de sécurité. Ainsi les auteurs de [45] n'ont pas proposé leur solution pour des courbes plus grandes que 256 bits, là où Guillermin propose des résultats d'implantations couvrant toutes les tailles standards du NIST (c'est à dire jusqu'à 521 bits).

Les compromis sur les choix de n (le nombre de canaux), et w (la taille d'un canal) n'ont pour le moment été que peu étudiés dans la littérature, mais, dans tous les cas, de telles études dépendraient très fortement de la plate-forme sur laquelle elles seraient menées. Un article très récent sur le choix de n est paru dans [126]. Dans ce papier est considérée une machine avec des mots de m bits (l'exemple de $m = 16$ est pris) et l'objectif est de compter les multiplications de m bits dans une multiplication modulaire RNS complète. Ici, m ne vaut pas forcément w (où w est toujours la taille des canaux). Dans leurs exemples, les meilleurs résultats suivant cette métrique montrent que les n optimaux, pour leurs modèles, sont assez petits pour une machine 16 bits (des bases de 4 moduli pour des corps de 256 ou 512 bits, 5 moduli pour 1024). Ce résultat met en évidence une certaine contradiction dans les calculs RNS. En prenant des bases avec plus d'éléments, c.-à-d. avec des w plus petits, on réduit le coût des multiplications. En effet, les multiplications sur les canaux `EMM` ont complexité de w^2, et les multiplications RNS ont une complexité de n en `EMM`, ce qui donne un coût de nw^2. Sachant qu'on a toujours, pour un corps fixé de taille

ℓ, $n \times w = \ell$, réduire w permet bien de réduire le coût total d'une multiplication RNS. En contrepartie, en réduisant w, on va augmenter le nombre d'opérations élémentaires pour faire nos réductions modulaires, il y a donc un équilibre à trouver entre les multiplications et les réductions modulaires.

Ceci dit, dans les faits, un tel décompte ne s'applique pas directement à une architecture parallèle. En effet, réduire le nombre de multiplications n'implique pas nécessairement de réduire le temps de calcul, le niveau de parallélisme est aussi un facteur très important. Notamment les multiplications sur beaucoup de canaux sont indépendantes et donc parallélisables, mais les sous-produits d'une multiplication pour un grand w le sont beaucoup moins. De plus, choisir un petit n impliquant un grand w, la chute de la fréquence des opérateurs arithmétiques doit être prise en compte, mais n'est pas chiffrable sans avoir implanté effectivement de telles unités. Si on prend par exemple les multiplieurs des constructeurs de FPGA qui sont très optimisés, il y aura à un moment donné un effet de seuil en terme de chute de fréquence dû à un routage devenant d'un coup plus compliqué. Et si ces grosses unités sont pipelinées pour augmenter la fréquence, outre le matériel supplémentaire requis, cela implique une complexification du contrôle pour utiliser efficacement ces unités. Enfin, d'un point de vue attaques physiques, les variations de consommation d'énergie de grosses unités de calcul risquent de se démarquer beaucoup plus sur les traces de consommation que des petits canaux. De même, l'architecture perd du coup en modularité si on considère peu d'unités très grosses. Pour toutes ces raisons, il est très difficile d'évaluer les véritables conséquences de ce type de métrique, le plus sûr (mais aussi de loin ce qui consomme le plus de temps) est d'effectuer un grand nombre d'implantations avec beaucoup de paramètres. Ce genre d'étude sera tôt ou tard nécessaire pour qui veut obtenir les meilleures performances dans une architecture RNS, même si une telle étude serait limitée à un ensemble de circuits (par exemple une famille de FPGA).

Architecture sélectionnée

Les travaux de cette thèse portent sur des propositions algorithmiques, arithmétiques implantées sur le modèle de l'architecture [52, 54]. L'architecture a été considérée comme une contrainte ici, bien que peut-être, une recherche architecturale améliorerait les résultats que nous avons obtenus avec nos nouveaux algorithmes. C'est pourquoi on détaille ici un peu plus les éléments internes de ce `Cox-Rower`, adapté de [64]. Les parties modifiées seront directement présentées dans les sections sur les contributions.

L'architecture implantée dans [52] (qu'on retrouve dans une version plus détaillée dans [54]) est très proche de celle de Kawamura *et al.*, avec un `Cox` et n `Rower`s (l'architecture est donc complètement parallélisée), mais sans la propagation de retenue. Les `Rower`s ne sont pas directement reliés en anneau dans cette architecture : les sorties sont ici reliées à un grand multiplexeur pour les rediriger vers tous les autres `Rower`s, lorsqu'on effectue l'extension de base (voir figure 1.7). Les `Rower`s ont aussi été modifiés pour pouvoir faire des opérations plus complexes que des carrés ou des multiplications par des constantes (ce qui ne posait pas de problèmes pour l'implantation RSA [64, 87]).

La figure 1.9, provenant de [54], présente l'unité arithmétique implantée dans cette contribution, avec 6 étages de pipeline. Sur la figure 1.9, ε correspond à $h_{a,i}$ dans les notations présentées précédemment. Cette unité a été spécialement conçue pour optimiser les calculs sur les courbes elliptiques et les réductions modulaires qu'ils requièrent. Cette

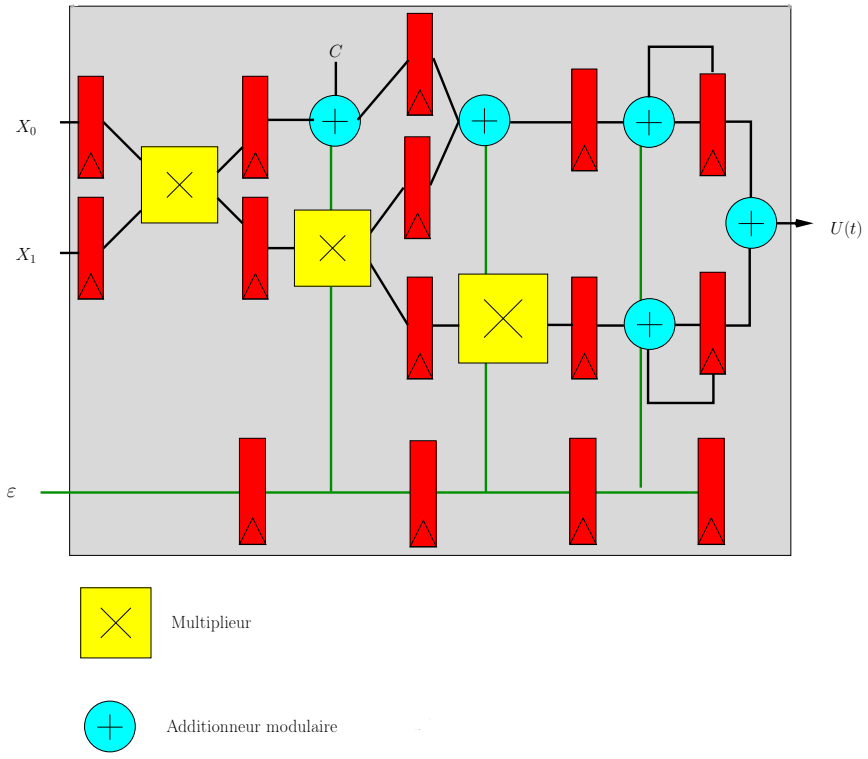

FIGURE 1.9 – Unité arithmétique contenue dans un `Rower` pour ECC (source [54]).

unité permet de calculer sur $\mathcal{B}_a$ la première base

$$U_t = |x_1 \cdot x_2 + b_0(-M_b) + b_1 U_{t-1}|_{m_{a,i}} \quad ,$$

où $b_0, b_1 \in \{0, 1\}$ et U_{t-1} est le résultat précédent (on définit de même les calculs sur $\mathcal{B}_b$). On rappelle que $h_{a,i}$ permet de déduire le canal sur lequel nous calculons. Cette unité nous permet donc de multiplier ses deux entrées, puis d'ajouter éventuellement au résultat $(-M_b)$ pour corriger l'extension de base avec la technique de Kawamura *et al.* [64]. Enfin, ce résultat peut être accumulé au résultat précédent, notamment afin de calculer efficacement la somme du CRT. On peut remarquer qu'avec une telle unité de calcul, une addition coûtera 2 cycles car il faudra calculer $x_1 \cdot 1 + x_2 \cdot 1$, alors qu'une multiplication ne prendra qu'un seul cycle. Par contre, les 2 cycles de l'addition seront transparents lorsque l'on fera des sommes de produits : faire $x_1 \cdot x_2 + x_3 \cdot x_4$ prendra autant de temps que de calculer indépendamment $x_1 \cdot x_2$ puis $x_3 \cdot x_4$. L'architecture est donc particulièrement efficace pour les sommes de produits (qui sont utilisées par les formules ECC pour le RNS par [8] ou encore pour le CRT).

À notre connaissance, très peu d'implantations FPGA d'ECC se sont révélées plus rapides que celle de Guillermin [52]. En dehors de la contribution RNS [45] dont nous avons parlé précédemment, les contributions [56] et [74] proposent des implantations (non RNS) ECC en FPGA plus rapides que [52]. En fait, il est toujours assez difficile de tirer de vrais conclusions autrement qu'en terme de performance pure. En effet, le papier de Güneysu et Paar [56] propose une implantation double-and-add, en utilisant les premiers du NIST P-224 et P-256 et les coordonnées de Chudnovsky obtenant une multiplication scalaire un

petit plus rapide (0.62 ms contre 0.68 ms pour 256 bits) que [52], mais en implantant une multiplication scalaire beaucoup moins coûteuse et avec un P très adapté à la représentation qu'ils utilisent. Enfin, à la différence de Guillermin qui a proposé une implantation générique pour 5 tailles du NIST, la proposition [56] est optimisée pour leur taille de corps. Ces faits sont juste exposés afin de justifier qu'il n'y a, pour le moment, pas de « meilleure » stratégie en terme de choix de l'arithmétique (surtout lorsque P est quelconque).

De même, la contribution de Ma *et al.* [74] propose cette fois-ci une implantation pour un P quelconque, avec une protection SPA, mais cette protection (celle de Möller [86]) réduit en fait beaucoup le nombre d'additions de points à effectuer (c'est un calcul fenêtré divisant par 4 le nombre d'additions de points par rapport à une échelle de Montgomery, pour les paramètres de fenêtre qu'ils ont choisis). On ne peut donc pas comparer équitablement les opérateurs arithmétiques proposés dans [74] et dans [52].

Pour conclure sur cette question, il y a plusieurs façons d'implanter ECC sur $\mathbb{F}_P$ avec des arithmétiques très différentes et de bonnes performances. De plus, des avancées autant sur l'arithmétique que sur l'utilisation des blocs FPGA étant régulièrement effectuées, il n'existe pour l'heure pas de meilleure solution.

Le RNS en tant que protection contre certaines attaques physiques

Les spécificités du RNS ont fait que plusieurs propositions de protection contre des attaques par canaux cachés ont été proposées. On a, premièrement, l'indépendance sur les canaux qui nous permet par exemple, d'effectuer les calculs d'un canal aléatoirement sur un `Rower` ou un autre, ou encore de rendre aléatoire l'ordre des calculs lorsque l'architecture n'est pas complètement parallélisée. Deuxièmement, d'autres choix d'introduction d'aléa sont possibles, comme le fait de choisir aléatoirement $\mathcal{B}_a$ parmi les $2n$ moduli nécessaires pour les réductions (ou plus), introduisant ainsi une représentation dans le domaine de Montgomery aléatoire (en effet, en RNS la valeur X, dans le domaine de Montgomery, est définie par XM_a).

Ainsi, la contribution [29] de Ciet *et al.* propose la randomisation des bases en tirant aléatoirement les $2n$ moduli dans un ensemble de moduli plus grand, avant de procéder à une exponentiation (logarithme discret ou RSA). Comme pointé dans [11], le calcul des constantes pour la réduction de Montgomery prend beaucoup de temps, et n'est pas envisageable (plusieurs To de pré-calculs, voir [11]). L'article [11] de Bajard *et al.* règle ce problème de pré-calculs, et propose d'effectuer le tirage aléatoire des bases soit avant l'exponentiation (ou multiplication scalaire), soit pendant l'exponentiation même, en utilisant simplement quelques appels à la multiplication de Montgomery RNS. Le terme de *leak resistant arithmetic* (LRA) est ainsi introduit pour désigner ces techniques. Dans le papier [53], Guillermin adapte la LRA pour les extensions de base de type CRT (dans [11] le MRS est utilisé), et les pré-calculs qui y sont associés. On y trouve aussi la première implantation de la LRA sur FPGA (menant à un surcoût de 15 % en surface et 6 % en temps). Enfin, l'article [94], de Perin *et al.*, propose une implantation du tirage aléatoire des bases avant exponentiation LRA et une implantation d'une autre contre mesure, la permutation des éléments internes à chacune des bases $\mathcal{B}_a$ et $\mathcal{B}_b$. Ces contre-mesures ont été ensuite attaquées par rayonnement électromagnétique afin d'évaluer leur sécurité pour ce type de canal caché. D'après leurs résultats, ces contre-mesures sont très peu coûteuses et améliorent significativement la résistance à certaines attaques sur le rayonnement électromagnétique.

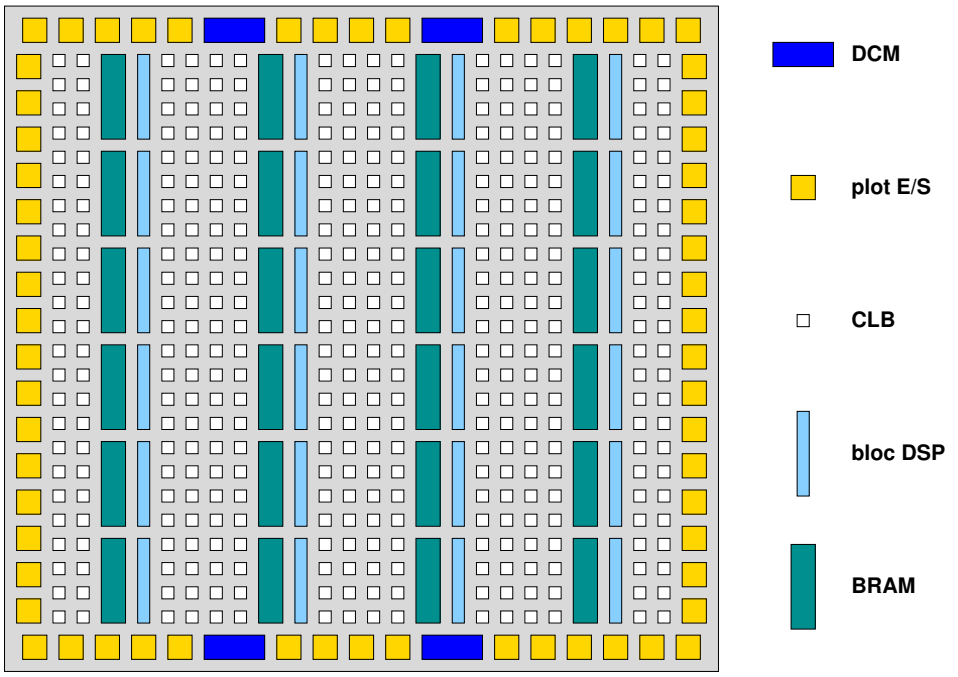

FIGURE 1.10 – Architecture générique d'un FPGA.

Enfin, des propositions utilisant un modulo supplémentaire ou le `Cox` ont été proposées pour protéger les implantations RNS contre certaines attaques de type injection de faute. Ciet *et al.* [29] proposent l'utilisation d'un modulo supplémentaire pour vérifier si, à la fin de l'exponentiation, le résultat n'est pas plus grand que ce qui est attendu (par exemple inférieur à $3P$). Guillermin [53] propose lui de détecter les fautes pour toutes les multiplications modulaires en transformant le `Cox`. La contribution de Bajard *et al.* [9] propose aussi une technique de détection de fautes pour chaque réduction modulaire, mais ne force pas l'utilisation d'une architecture `Cox-Rower`. De plus, un modèle de fautes plus fin est utilisé suivant le moment où la faute se produit.

Les contributions de cette thèse portant surtout sur l'amélioration de l'efficacité des calculs en RNS, ces questions ne seront pas plus approfondies. On notera toutefois que ces contre-mesures ne sont qu'un arsenal de protections supplémentaires, en plus des protections au niveau de la multiplication scalaire comme celles de Coron [33] (randomisation du point de base en coordonnées projectives, masquage du point de base ou masquage de la clé) ou des protections matérielles comme des logiques avec moins de variation de consommation de courant [76, 122].

À propos des circuits FPGA

Les implantations réalisées dans cette thèse ont été effectuées sur FPGA (pour *field programmable gate array*), comme bon nombre d'implantations cryptographiques de la littérature, notamment en RNS comme le montre la table 1.6. Dans cette dernière section consacrée aux implantations, nous rappelons ici quelques informations sur les FPGA.

Les FPGA sont un type de circuit intégré complètement programmable après leur fabrication (ils ont une architecture dite configurable et souvent reconfigurable un grand

nombre de fois). Ces technologies sont très avancées (finesse de gravure de 28 nm pour les Virtex 7 et Stratix V, introduits en 2010), et sont beaucoup plus accessibles que les ASIC (*application-specific integrated circuit*) qui demandent un énorme investissement dans la conception et la fabrication. Les délais de fabrication chez le fondeur pouvant être importants (plusieurs mois), la conception et le test sur FPGA sont beaucoup plus rapides à effectuer, et nécessitent beaucoup moins de personnel. Ces circuits sont fabriqués en grand volume et coûtent de quelques euros à plusieurs milliers d'euros. Ces prix font que les FPGA sont plus rentables que les ASIC pour les petites séries, les circuits ASIC ne deviennent rentables que pour de très grands volumes. Les FPGA sont aussi utilisés pour faire du prototypage.

Les FPGA offrent donc une possibilité de mise sur le marché rapide, mais, en contrepartie, consomment plus d'énergie que les ASIC, pour des performances moins élevées. Ceci est dû aux mécanismes permettant la configuration du FPGA. Enfin, ces produits fournissent les très bonnes performances du matériel avec les avantages d'une configuration logicielle, on les retrouve donc dans de nombreuses applications (calcul hautes performances, réseaux sans fil, imagerie médicale, dispositifs de sécurité, etc). Une bonne introduction aux FPGA peut être trouvée dans [111].

Un FPGA peut être vu comme un tableau de portes logiques (d'où son nom). Une mémoire va être alors directement programmée par l'utilisateur, qui va définir les fonctionnalités des différents blocs ainsi que leurs interconnexions. La figure 1.10 présente l'architecture générique d'un FPGA. On y voit une large matrice composée de blocs logiques configurables (CLB pour *configurable logic bloc*), de blocs d'entrées/sorties et de blocs de génération d'horloge programmable (DCM pour *digital clock manager*). Prenons l'exemple des FPGA Xilinx Virtex 5, sur lesquels les implantations de cette thèse ont été faites. Sur Virtex 5, un CLB est composé de deux *slices*, qui sont les blocs de base des FPGA Xilinx. Pour cette famille de FPGA, chaque *slice* est composé de 4 LUT à 6 entrées (ou table de correspondance, *look-up table* en anglais) et de 4 bascules. Très souvent, d'autres types de blocs spécifiques cassent l'homogénéité d'une telle architecture pour gagner en performance ou en consommation d'énergie. Par exemple, on trouve fréquemment des BRAM, blocs de mémoire RAM ou encore des blocs arithmétiques optimisés (généralement optimisés pour les calculs de traitement du signal, appelés blocs DSP). Chez Xilinx par exemple, les blocs DSP sur Virtex 5 contiennent un multiplieur-accumulateur, permettant d'effectuer des multiplications 18×25 bits et des les accumuler sur 48 bits, ou d'effectuer un simple multiplication ou addition (voir figure 1.11, issue de [2], page 14). Un exemple provenant du concurrent Altera est lui présenté dans la figure 1.12 issue de [1] (chapitre 7, page 5). Ce bloc là, présent dans les Stratix 3, contient 4 multiplieurs 18×18 bits sur chacune de ses moitiés, et toujours des fonctions d'additions et d'accumulations. Les `Rower`s vont en fait combiner ces blocs dans l'unité arithmétique qu'ils intègrent.

Par contre, la possibilité de pouvoir configurer et reconfigurer tous ces blocs a un coût. Tous les composants d'interconnexion (ces nœuds ne sont pas représentés sur la figure 1.10) sont ajoutés par rapport à un circuit non reconfigurable. Ceux-ci impliquent une perte de performances (en énergie et fréquence d'horloge) ainsi qu'une perte de surface. D'après [111], la surface totale occupée pour ce routage et la mémoire de configuration représente 90 % de la surface du FPGA. De plus, les fréquences de fonctionnement des FPGA ne dépassent pas quelques centaines de MHz, contre les GHz des microprocesseurs. Par contre, les FPGA peuvent contrebalancer cette différence de fréquence et être plus

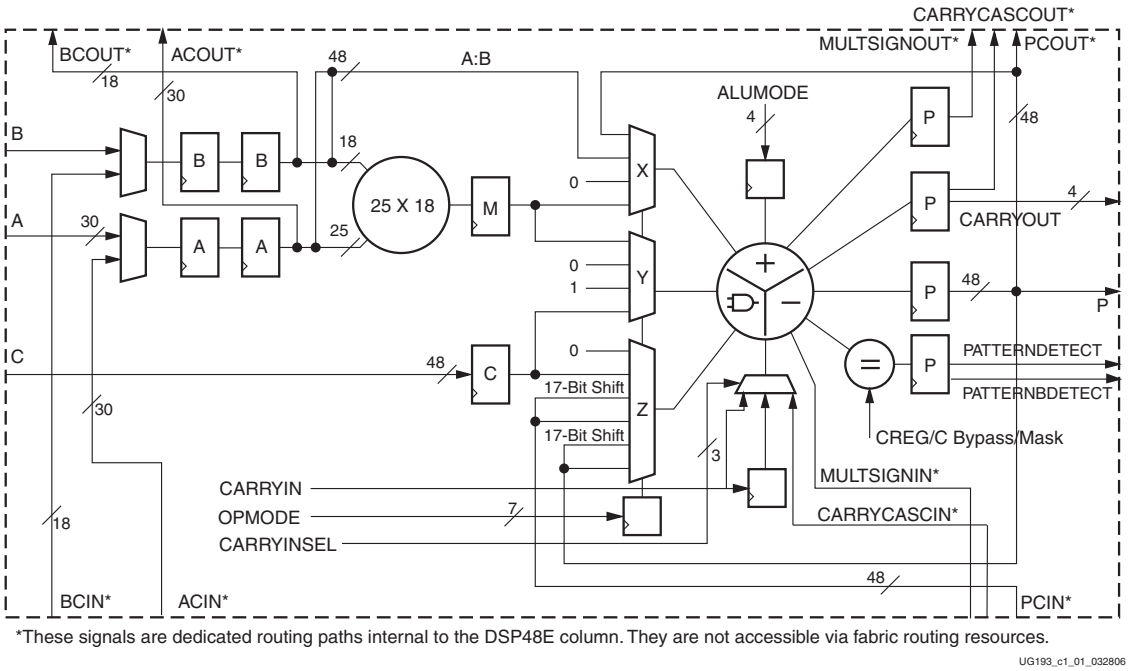

FIGURE 1.11 – Schéma de bloc DSP Virtex 5 issu de [2].

rapides pour de nombreuses applications grâce au fait qu'ils permettent une implantation spécifique à l'application, souvent parallèle, et à grain fin (architecture au bit près). Les FPGA sont souvent présentés comme une solution intermédiaire entre une solution purement logicielle, peu chère en développement mais plus faible en performances, et une solution de type ASIC, très chère en développement mais très performante.

La programmation d'un FPGA se fait par un grand nombre de configurations de mémoires pour effectuer la fonction souhaitée. Cette configuration va permettre de programmer les LUT, les blocs arithmétiques (DSP), l'initialisation des mémoires et les connexions qui relient tous ces composants. Le concepteur va en fait utiliser des outils de CAO qui permettront de transformer une implantation en langage de description matérielle (HDL) en un flot binaire appelé *bitsream* qui sera chargé dans le FPGA pour le programmer. Dans cette thèse, les travaux d'implantation ont été effectués en langage VHDL. Ce langage permet de décrire au niveau RTL (*register transfer level*) l'architecture en cours de conception.

Autrement dit, on programme en utilisant des fonctions combinatoires de base (fonctions arithmétiques, logiques, multiplexeur, etc), des fonctions synchrones (la mémoire, les registres, les bascules) et en décrivant les signaux qui les relient. Ce code est ensuite synthétisé par un outil tel que l'outil ISE de Xilinx ou Quartus d'Altera. Il sera analysé par l'outil qui extraira les blocs arithmétiques pour utiliser les ressources adéquates (avec une utilisation, ou non des blocs DSP par exemple), puis simplifiera les équations logiques suivant les options d'optimisation activées (optimisation temps ou surface). Cette analyse effectuée, l'outil passera à une étape de transformation technologique afin de faire correspondre les blocs à des instances matérielles existantes sur le FPGA, en essayant de tenir

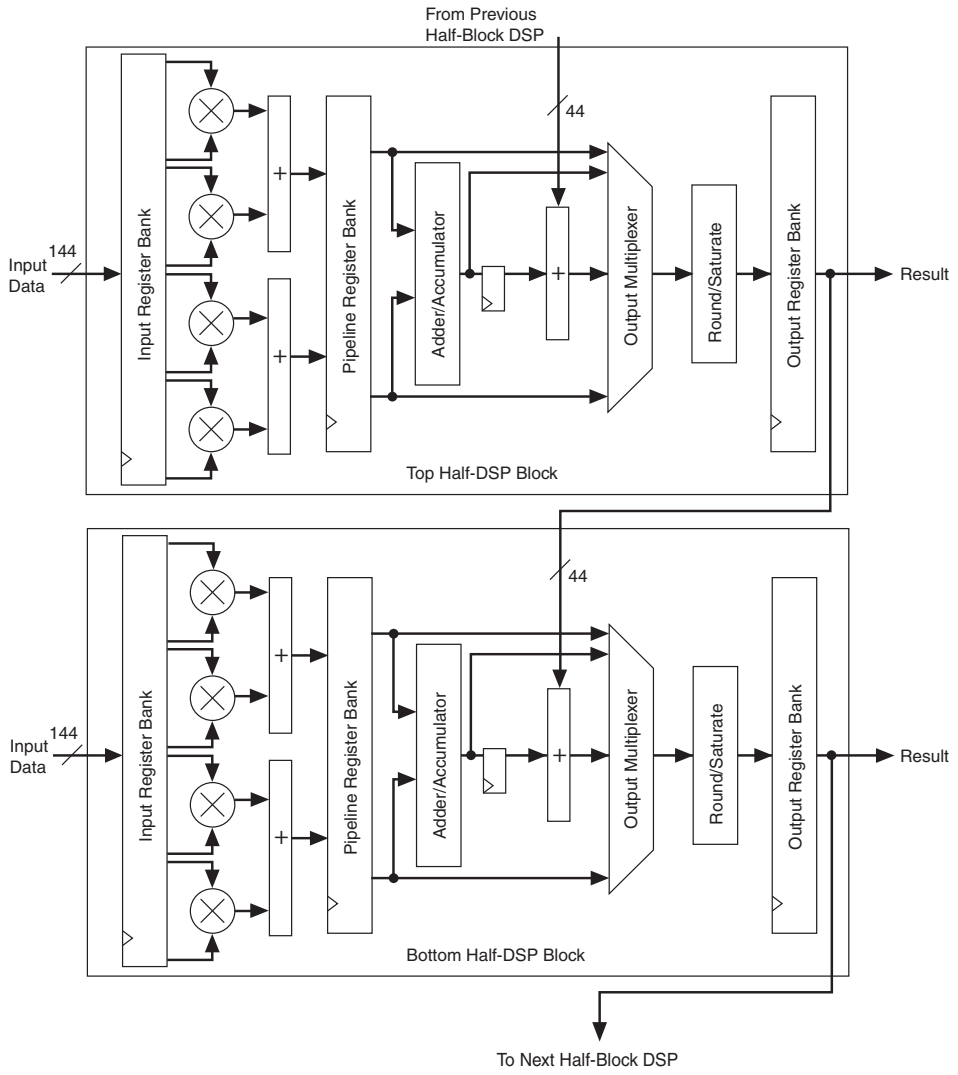

FIGURE 1.12 – Schéma de bloc DSP Stratix III issue de [1].

compte au mieux des contraintes imposées. Enfin le placement et le routage vont placer ces unités sur les blocs du FPGA et les relier en utilisant les différentes ressources de routage. Afin de simuler le comportement du code VHDL implanté, du code VHDL spécifique peut être utilisé (du code non synthétisable) pour écrire un banc d'essais. Le comportement est alors simulé par des logiciels tels que ModelSim ou Isim.

Chapitre 2

Inversion modulaire rapide en RNS

Dans certaines implantations matérielles cryptographiques utilisant le RNS, comme ECC [45, 52, 104] ou encore les couplages [27, 127] une ou des inversions modulaires sont nécessaires. Par exemple, dans le cas des courbes elliptiques, nous avons besoin d'au moins une inversion finale afin de normaliser en affine les coordonnées du point obtenu lors de la multiplication scalaire. Les unités de calcul modulaire sur de grands nombres étant implantées en RNS à des fins de performances, on les utilise aussi pour faire cette ou ces inversions. Cependant cette opération reste très coûteuse en RNS, même devant le coût d'une multiplication scalaire. En effet, nous allons voir que nous pouvons estimer que dans le cas du RNS, l'inversion coûte entre 10 et 20% du coût de la multiplication scalaire complète. Dans le cas des couplages, il est dit par exemple dans [27] « *the remaining inversion in* $\mathbb{F}_p$ *is very expensive* » à propos de l'inversion RNS. Dans ce chapitre sont présentés les résultats publiés dans [19], ainsi qu'une extension en cours de soumission à un journal. Un nouvel algorithme d'inversion RNS est présenté, permettant de réduire très significativement le nombre d'opérations élémentaires, aussi bien que le temps d'exécution sur nos implantations FPGA en section 2.2. Ensuite, une variante de cet algorithme menant à une réduction de 30 % de `EMM` (multiplication modulaire élémentaire de w bits) et de 20 % de `EMA` (addition modulaire élémentaire) est présentée en section 2.3. Les sections suivantes proposent une comparaison avec l'état de l'art en section 2.4 ainsi que les détails d'implantations et les résultats obtenus en section 2.5.

2.1 Inversion modulaire RNS dans l'état de l'art

Dans quasiment toutes les implantations RNS de l'état de l'art (en tout cas en matériel), l'algorithme utilisé pour calculer l'inversion modulaire est une adaptation en RNS du calcul de l'exponentiation utilisant le petit théorème de Fermat. L'algorithme 16 présente une façon de faire cette inversion, que nous avons ici modifiée en une version *bit de poids faible en premier* (LSBF pour *least significant bit first*) de l'exponentiation RNS de l'état de l'art [48]. On rappelle que le but de l'algorithme est de calculer $A^{P-2} \bmod P$ ce qui est égal à $A^{-1} \bmod P$ d'après le FLT. L'algorithme met en œuvre les optimisations présentées dans la section 1.3.3 provenant de [48], on se place dans la représentation RNS modifiée sur la seconde base (ligne 1 de l'algorithme 16) puis on reconvertit à la fin en multipliant par $\overrightarrow{T_b}$ (ligne 9). Cet algorithme, grâce à sa représentation RNS modifiée, permet d'être plus rapide en accélérant `MM`, comme expliqué dans la section 1.3.3. Mettre cet algorithme en version LSBF permet d'avoir la multiplication (ligne 5) et le carré (ligne 6) exécutés en parallèle. Cet algorithme sera utilisé par la suite afin de se comparer à l'état de l'art.

Un défaut de cet algorithme d'inversion est qu'il y a beaucoup de dépendances entre les itérations de la boucle (comme dans tout algorithme d'exponentiation), ce qui limite fortement le remplissage du pipeline. Dans la suite, cette inversion modulaire RNS sera notée `FLT-MI`. Les auteurs de [127], qui ont obtenu parmi les meilleures performances pour une implantation FPGA de couplages et qui ont implanté `FLT-MI` (dans un algorithme très proche de l'algorithme 16), appuient ce constat en écrivant :

« *The problem with this exponentiation in $\mathbb{F}_p$ is that the computation cannot be pipelined [...] causes low pipeline occupation rate and a huge waste.* »

Une exponentiation comme l'algorithme 16 représente environ 10 à 20% du coût d'une multiplication scalaire complète sur courbe elliptique (en terme d'effort de calcul). En effet, on peut approcher le coût global en ne considérant que les réductions modulaires, tant le déséquilibre est important en RNS entre cette opération et les autres opérations de base. Ainsi, `FLT-MI` coûte environ $\log_2 P \times 1.5$ réductions modulaires. Dans la contribution [52], des formules adaptées pour le RNS sont utilisées avec moins de réductions modulaires, et avec l'algorithme de multiplication scalaire « échelle de Montgomery » (voir algorithme 22 protégé contre la SPA dans la section 3.2.1 par exemple). Pour chaque bit de clé, une addition et un doublement de points sont effectués, ce qui coûte 13 réductions modulaires. On a donc l'inversion qui coûte $\log_2 P \times 1.5$ contre $\log_2 P \times 13$ pour la multiplication scalaire, soit, ramené au système global « multiplication scalaire suivie d'une inversion », l'inversion coûte plus de 10% juste en terme d'opérations. Ce ratio peut même être plus élevé suivant le choix des formules ou de l'algorithme de multiplication scalaire. De plus, l'auteur de [52] indique que la multiplication scalaire ne contient pas ou quasiment pas de cycle d'attente (chaque `ADD` et `DBL` contient certaines opérations indépendantes, et l'utilisation d'une version LSBF de l'échelle de Montgomery fait que l'on peut réaliser en parallèle une `ADD` et un `DBL`). Comme déjà indiqué, l'inversion `FLT-MI` contient beaucoup de cycles d'attente, on peut donc estimer que l'inversion prend bien plus de 10% du temps total d'exécution de la séquence multiplication scalaire/inversion. Malgré tout, cet algorithme a l'avantage d'être très régulier et de n'utiliser que des ressources matérielles déjà présentes et déjà optimisées pour les autres calculs. De plus, le fait d'effectuer l'opération en RNS permet de ne pas avoir d'un côté tout le matériel dédié au calcul RNS inactif pendant l'inversion, et de l'autre de gros opérateurs arithmétiques de plusieurs centaines de bits en représentation standard juste pour le temps de l'inversion.

En dehors de l'algorithme `FLT-MI`, peu d'alternatives s'offrent à ceux qui désirent implanter l'inversion modulaire en RNS. Les autres solutions dont nous disposons dans le cas standard, c'est à dire l'algorithme d'Euclide étendu et sa version binaire, ont souvent été mises de côté du fait de la difficulté des opérations telles que la division ou la comparaison, comme en témoigne par exemple N. Guillermin dans sa thèse [54], où il écrit :

« *En effet, la comparaison étant chère en RNS, l'inversion ne peut pas être réalisée par le classique algorithme d'Euclide étendu, mais par une exponentiation onéreuse par $p - 2$* ».

Une alternative a été proposée dans [13], se basant sur l'algorithme d'Euclide. Les auteurs proposent une façon d'obtenir une approximation U_3/V_3 du quotient ligne 5 de l'algorithme 10, à partir d'approximations de U_3/M et V_3/M (M est le produit des moduli de la base). Malheureusement, cet algorithme n'a, à notre connaissance, jamais été implanté et sa complexité jamais évaluée. En effet, la fonction de base de l'algorithme permettant l'approximation de U_3/M, appelée *ApproxSup*, n'a pas été évaluée. Cependant, elle est composée d'une accumulation de produits, qui est donc peu parallélisable,

Algorithme 16: Inversion modulaire `FLT-MI` basée sur le petit théorème de Fermat (version LSBF de [48]).

Entrées : $\overrightarrow{A_{a|b}}$, $P-2=(1\,p_{\ell-2}\dots p_0)_2$
Pré-calculs : P, $\overrightarrow{(|M_a|_P)_a}$, $\overrightarrow{(|M_a|_P T_b^{-1})_b}$, $\overrightarrow{(|M_a^2|_P)_a}$, $\overrightarrow{(|M_a^2|_P T_b^{-1})_b}$, $\overrightarrow{T_b}$, $\overrightarrow{(T_b)^{-1}}$
Sortie : $\overrightarrow{S_{a|b}} = \overrightarrow{(|A^{P-2}|_P)_{a|b}}$

1 $\overrightarrow{R_{a|b}} \leftarrow (\overrightarrow{A_a}, \overrightarrow{(A_b \cdot T_b^{-1})_b})$
2 $\overrightarrow{R_{a|b}} \leftarrow \texttt{MM}\left(\overrightarrow{R_a}, \overrightarrow{R_b}, \overrightarrow{(|M_a^2|_P)_a}, \overrightarrow{(|M_a^2|_P T_b^{-1})_b}\right)$
3 $\overrightarrow{S_{a|b}} \leftarrow (\overrightarrow{(|M_a|_P)_a}, \overrightarrow{(|M_a|_P T_b)_b})$
4 **pour** i **de** 0 **à** $\ell-2$ **faire**
5 **si** $p_i = 1$ **alors** $\overrightarrow{S_{a|b}} \leftarrow \texttt{MM}(\overrightarrow{S_a}, \overrightarrow{S_b}, \overrightarrow{R_a}, \overrightarrow{R_b})$
6 $\overrightarrow{R_{a|b}} \leftarrow \texttt{MM}(\overrightarrow{R_a}, \overrightarrow{R_b}, \overrightarrow{R_a}, \overrightarrow{R_b})$
7 $\overrightarrow{S_{a|b}} \leftarrow \texttt{MM}(\overrightarrow{S_a}, \overrightarrow{S_b}, \overrightarrow{R_a}, \overrightarrow{R_b})$
8 $\overrightarrow{S_{a|b}} \leftarrow \texttt{MM}(\overrightarrow{S_a}, \overrightarrow{S_b}, \overrightarrow{1_a}, \overrightarrow{(T_b)^{-1}})$
9 $\overrightarrow{S_b} \leftarrow \overrightarrow{S_b \cdot T_b}$
10 **retourner** $\overrightarrow{S_{a|b}}$

notamment peu adaptée à une architecture complètement parallélisée comme souvent le cas dans les architectures RNS pour la cryptographie. De plus, cette accumulation est faite sur des mots de $2w$ bits, c'est à dire la taille de deux moduli (ce qui implique une unité spécialisée, ou alors créer des paires de modulo capables d'effectuer des opérations sur $2w$ bits). Une autre fonction, appelée *NormalSup*, fait ensuite plusieurs appels à *ApproxSup*. Ce nombre d'appels n'est pas clairement identifié dans la contribution (l'appel s'effectue dans une boucle *while*). Enfin, l'impact de telles approximations sur le nombre de tours de boucle principale de l'algorithme d'Euclide étendu n'a pas été évalué, les auteurs expliquant en conclusion qu'une évaluation de l'algorithme est nécessaire. Pour conclure, ces raisons laissent penser qu'une implantation de cet algorithme serait coûteuse, notamment en terme de temps d'exécution sur une architecture parallèle et en terme de matériel supplémentaire.

Enfin, l'article [72] propose une implantation sur microprocesseur d'ECC en RNS. Dans cette contribution est évoquée une adaptation RNS de la variante binaire d'Euclide (algorithme 11). Aucun détail n'est fourni sur cette adaptation logicielle, hormis le fait qu'elle résulte en une inversion presque 2 fois plus rapide (réduction du temps d'exécution de 48%).

Pour conclure sur cet état de l'art, l'algorithme de Fermat propose en matériel un algorithme facile à implanter, réutilisant directement les opérateurs présents, à la différence d'une implantation matérielle d'un algorithme d'Euclide étendu ou sa variante binaire (par exemple pour les comparaisons). De plus, ces algorithmes sont en $O(\log_2 P)$ (c.-à-d. $O(\ell)$) itérations de la boucle principale, la différence se situe surtout dans le coût d'une itération de cette boucle. Cette différence ne semble pas forcément en défaveur du `FLT-MI` à cause du coût des comparaisons ou divisions en RNS. Pour toutes ces raisons, la seule vraie proposition matérielle de l'état de l'art est l'implantation du `FLT-MI`.

2.2 Inversion modulaire plus-minus en RNS

Notre algorithme d'inversion modulaire en RNS est basé sur l'algorithme d'Euclide étendu en version binaire, avec l'utilisation de l'astuce plus-minus [22] (présentée section 1.2.3). Cette astuce va nous permettre d'éviter les comparaisons sur les valeurs RNS. L'algorithme obtenu est significativement plus rapide que les inversions basées sur le FLT. De plus, notre algorithme, comme les solutions basées sur le FLT, va principalement réutiliser les ressources matérielles qui sont requises pour les calculs sur les corps finis, comme l'addition, la soustraction ou la multiplication, menant à un très faible surcoût en surface, et donc à une haute efficacité architecturale ; il n'y a pas de grosses unités dédiées inactives pour les autres calculs. Nous allons voir que notre proposition ne nécessite qu'une seule base RNS, au lieu de deux dans l'état de l'art. Pour simplifier les notations, on considérera parfois de façon implicite que les valeurs sont dans la base $\mathcal{B}_a$. L'algorithme 17 a été publié dans [19] (conférence CHES 2013).

Le but de l'algorithme d'Euclide étendu binaire est d'utiliser le fait que l'on soit capable d'effectuer des divisions par 2 et des modulo 2 très efficacement (vu que l'on va utiliser le plus-minus, il faut savoir faire de même pour 4). Si c'est évident en représentation binaire de position, ceci l'est beaucoup moins dans la représentation RNS. En RNS, on peut se placer dans 3 stratégies.

Première stratégie, on choisit un élément m_γ premier avec la base $\mathcal{B}_a$ tel qu'il soit un multiple de 4 (par exemple $m_\gamma = 2^w$ avec $w > 1$). Ainsi, la réduction modulo 4 devient très simple : il suffit de tester les 2 derniers bits du reste de la valeur testée en m_γ. Cependant, dans ce cas, on ne peut pas diviser par 2 ou 4 modulo m_γ. Pour récupérer le résultat d'une division par 4, on réalise d'abord la division dans la base $\mathcal{B}_a$, qui est choisie impaire, en effectuant une multiplication par 4^{-1} mod M_a. Ensuite, on envoie le résultat dans m_γ grâce à une extension de base à partir de $\mathcal{B}_a$. On doit donc calculer la formule du CRT pour m_γ :

$$|X|_{m_\gamma} = \left| \sum_{i=1}^{n} \xi_{a,i} \cdot \left| \frac{M_a}{m_{a,i}} \right|_{m_\gamma} - |q \cdot M_a|_{m_\gamma} \right|_{m_\gamma}.$$

Cette accumulation là n'est pas parallèle, elle prendra donc n cycles sur un `Rower` (bloc d'arithmétique modulaire de w bits, voir la figure 1.9 section 1.3.5). On peut paralléliser une partie de ces cycles avec les calculs sur la base $\mathcal{B}_a$ si un `Rower` dédié au modulo m_γ est rajouté (engendrant un `Rower` qui ne sera utilisé que pour l'inversion). L'autre possibilité est de réserver un `Rower` d'un autre modulo lorsqu'on en a besoin, ce qui veut dire que toute l'architecture attend ces n cycles dans le cas d'une architecture complètement parallèle. Par soucis de simplicité du contrôle, on préfère garder les calculs sur les différents canaux de la base $\mathcal{B}_a$ synchronisés.

La deuxième stratégie est de choisir directement un élément de $\mathcal{B}_a$, par exemple $m_{a,1}$, égal à 2^w. L'idée est que lorsque l'on calcule $\overrightarrow{U}/2$ dans $\mathcal{B}_a$, on va faire notre multiplication par l'inverse de 2 dans tous les moduli de $\mathcal{B}_a$ sauf $m_{a,1}$, dans lequel on va juste faire un décalage. Ce décalage est une division par 2 du reste de U modulo 2, mais n'est pas une division par 2 modulo 2^w. Pour bien expliquer le phénomène, posons U un entier pair et

Algorithme 17: Inversion modulaire plus-minus proposée `PM-MI` (version binaire).

Entrées : $\overrightarrow{A}, P > 2$ avec $\text{pgcd}(A, P) = 1$
Pré-calculs : $\overrightarrow{C}, \overrightarrow{C/2}, \overrightarrow{(3\,C/4)}, \overrightarrow{(PT_a^{-1})/4}, \overrightarrow{(-PT_a^{-1})/4}, \overrightarrow{(PT_a^{-1})/2}, \overrightarrow{T_a}, \overrightarrow{T_a^{-1}}, |P|_4$
Sortie : $\overrightarrow{S} = \overrightarrow{|A^{-1}|_P}$, $0 \leqslant S < 2\,P$

$l_u \leftarrow 0$, $\widehat{U_1} \leftarrow \overrightarrow{C}$ /* $\widehat{U_1} = \widehat{0}$ */
$\widehat{U_3} \leftarrow \overrightarrow{P} \times \overrightarrow{T_a^{-1}} + \overrightarrow{C}$ /* $\widehat{U_3} = \widehat{P}$*/
$b_{U_1} \leftarrow 0$, $b_{U_3} \leftarrow |P|_4$
$l_v \leftarrow 0$, $\widehat{V_1} \leftarrow \overrightarrow{T_a^{-1}} + \overrightarrow{C}$ /* $\widehat{V_1} = \widehat{1}$ */
$\widehat{V_3} \leftarrow \overrightarrow{A} \times \overrightarrow{T_a^{-1}} + \overrightarrow{C}$ /* $\widehat{V_3} = \widehat{A}$ */
$b_{V_1} \leftarrow 1$, $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$
tant que $\widehat{V_3} \neq \widehat{1}$ *and* $\widehat{U_3} \neq \widehat{1}$ *and* $\widehat{V_3} \neq \widehat{-1}$ *and* $\widehat{U_3} \neq \widehat{-1}$ **faire**
  tant que $|b_{V_3}|_2 = 0$ **faire**
    si $b_{V_3} = 0$ **alors** $r \leftarrow 2$ **sinon** $r \leftarrow 1$
    $\widehat{V_3} \leftarrow \widehat{\texttt{div2r}}(\widehat{V_3}, r, b_{V_3})$
    $\widehat{V_1} \leftarrow \widehat{\texttt{div2r}}(\widehat{V_1}, r, b_{V_1})$
    $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$, $l_v \leftarrow l_v + r$
  $\widehat{V_3^*} \leftarrow \widehat{V_3}$, $\widehat{V_1^*} \leftarrow \widehat{V_1}$
  si $|b_{V_3} + b_{U_3}|_4 = 0$ **alors**
    $\widehat{V_3} \leftarrow \widehat{\texttt{div2r}}(\widehat{V_3} + \widehat{U_3} - \overrightarrow{C}, 2, 0)$,
    $\widehat{V_1} \leftarrow \widehat{\texttt{div2r}}(\widehat{V_1} + \widehat{U_1} - \overrightarrow{C}, 2, |b_{V_1} + b_{U_1}|_4)$
  sinon
    $\widehat{V_3} \leftarrow \widehat{\texttt{div2r}}(\widehat{V_3} - \widehat{U_3} + \overrightarrow{C}, 2, 0)$,
    $\widehat{V_1} \leftarrow \widehat{\texttt{div2r}}(\widehat{V_1} - \widehat{U_1} + \overrightarrow{C}, 2, |b_{V_1} - b_{U_1}|_4)$
  si $l_v > l_u$ **alors**
    $\widehat{U_3} \leftarrow \widehat{V_3^*}$, $b_{U_3} \leftarrow b_{V_3}$
    $\widehat{U_1} \leftarrow \widehat{V_1^*}$, $b_{U_1} \leftarrow b_{V_1}$
    $\text{swap}(l_u, l_v)$
  $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$
  $l_v \leftarrow l_v + 1$
si $\widehat{V_3} = \widehat{1}$ **alors retourner** $(\widehat{V_1} - \overrightarrow{C})\overrightarrow{T_a} + \overrightarrow{P}$
sinon si $\widehat{U_3} = \widehat{1}$ **alors retourner** $(\widehat{U_1} - \overrightarrow{C})\overrightarrow{T_a} + \overrightarrow{P}$
sinon si $\widehat{V_3} = \widehat{-1}$ **alors retourner** $-(\widehat{V_1} - \overrightarrow{C})\overrightarrow{T_a} + \overrightarrow{P}$
sinon retourner $-(\widehat{U_1} - \overrightarrow{C})\overrightarrow{T_a} + \overrightarrow{P}$

posons K_u et r_u tels que $U = K_u m_{a,1} + r_u = K_u 2^w + r_u$. Nous obtenons :

$$\begin{cases} \dfrac{U}{2} = \dfrac{K_u}{2}\, 2^w + \dfrac{r_u}{2} = \dfrac{K_u}{2}\, m_{a,1} + \dfrac{r_u}{2} & \text{si } K_u \text{ pair}, \\ \dfrac{U}{2} = K_u\, 2^{w-1} + \dfrac{r_u}{2} = K_u\, \dfrac{m_{a,1}}{2} + \dfrac{r_u}{2} & \text{si } K_u \text{ impair}, \end{cases} \tag{2.1}$$

si nous divisons U par 2. Si K_u est pair alors on a bien $\frac{r_u}{2} = |\frac{U}{2}|_{m_{a,1}}$. Par contre, si K_u est impair $\frac{r_u}{2} = |\frac{U}{2}|_{m_{a,1}/2}$, il nous manque le bit de poids fort. Ce n'est pas forcément un problème car si U est représentable dans $\mathcal{B}_a$ (c'est à dire pas de dépassement de la représentation en base $\mathcal{B}_a$), alors $U/2$ est représentable dans cette nouvelle base $\mathcal{B}_a = (m_{a,1}/2, m_{a,2}, \ldots, m_{a,n})$. À chaque division par 2, le premier modulo de la base $\mathcal{B}_a$ sera ainsi « divisé par 2 », jusqu'à ce que $m_{a,1} = 1$. Calculer K_u pour chaque division coûterait trop cher, c'est pourquoi il faut diviser par 2 la valeur $m_{a,1}$ pour considérer tout le temps le pire cas (impair) de l'équation 2.1. Lorsque $m_{a,1} = 1$, on doit faire une extension de base à partir des moduli « fixes » de la base $\mathcal{B}_a$, c'est à dire $(m_{a,2}, \ldots, m_{a,n})$, vers 2^w, ce qui permet de remettre $m_{a,1}$ à sa valeur initiale. On peut remarquer qu'en réalité, si on divise par 2 une certaine valeur U et par une autre V alors ces 2 valeurs ne seront pas représentées tout à fait sur la même base : le premier élément de $\mathcal{B}_a$ sera $m_{a,1}/2$ pour l'un et $m_{a,1}$ pour l'autre. Il faut donc associer un compteur à chaque donnée traitée dans l'algorithme afin de savoir exactement dans quelle base elle est représentée. L'avantage de cette deuxième solution par rapport à la première vient du fait qu'elle nécessite beaucoup moins d'extensions de base vers le modulo valant 2^w. De plus, le modulo 2^w est directement inclus dans la base $\mathcal{B}_a$, et n'est pas un modulo supplémentaire. Par contre, cette solution impose un contrôle complexe pour gérer le fait que $m_{a,1}$ change suivant le nombre de divisions par 2. De plus, dans un algorithme d'Euclide étendu binaire, il faut effectuer des opérations du type $(U + V)/2$. Pour pouvoir faire la somme, il faut que U et V soient représentés de la même façon (c'est à dire après le même nombre de divisions par 2). Si ce n'est pas le cas, il faut que celui qui ait été le plus été divisé, disons U, soit étendu pour être représenté dans la même base que V. Cette extension arrive alors avant qu'on ait $m_{a,1} = 1$ pour la valeur U, elle est en quelque sorte « prématurée » car l'extension de base aurait pu arriver plus tard s'il n'y avait pas eu le calcul de l'opération $(U + V)/2$.

Pour finir, la troisième stratégie est celle que nous avons retenue. Elle consiste à choisir une base sans aucun modulo pair, ainsi les divisions (exactes) par 2 et 4 sont très faciles. En contrepartie, il faut trouver un moyen d'obtenir efficacement nos valeurs modulo 2 et 4. L'algorithme 17 présente la méthode proposée. Avant de définir précisément les différentes fonctions et notations de l'algorithme, on peut remarquer tout de suite que l'on retrouve la structure de l'algorithme d'Euclide étendu binaire, avec le test plus-minus modulo 4 en ligne 14.

Tout d'abord, notre algorithme prend en entrée la valeur à inverser $\overrightarrow{A}$, et retourne l'inverse de cette valeur dans $[0, 2P]$ (comme pour les valeurs de sortie des multiplications de Montgomery). Il faut noter que notre inversion ne tient pas compte du fait que l'entrée est en représentation de Montgomery ou non. Si c'est le cas, il suffit de multiplier notre entrée par M_a^{-2} modulo P avant d'appliquer notre algorithme, pour obtenir l'inverse dans la représentation de Montgomery. Ensuite, l'algorithme utilise des valeurs dans une représentation $\widehat{X}$, que l'on peut voir comme une fonction affine appliquée à la représentation RNS habituelle de $\overrightarrow{X}$. Celle-ci sera définie dans les détails sur les fonctions `mod4` et $\widehat{\texttt{div2r}}$.

Avant le début de la boucle principale, on retrouve la même initialisation que pour l'algorithme 11 dans sa version RNS, avec quelques petits rajouts. Les variables l_u et l_v vont permettre de compter le nombre de fois que U_3 et V_3 respectivement ont été divisés par 2. C'est ce qui permet aux méthodes utilisant l'astuce plus-minus de faire en sorte que l'algorithme converge bien. Ensuite, les valeurs b_{U_1}, b_{U_3}, b_{V_1} et b_{V_3} stockent tout simplement leur valeur en indice modulo 4.

La boucle principale se termine lorsque l'une des valeurs $\widehat{U_3}$ ou $\widehat{V_3}$ vaut 1 en valeur absolue. Comme pour l'algorithme d'Euclide étendu binaire, on a toujours $|U_1 A|_P = U_3$ et $|V_1 A|_P = V_3$, donc si par exemple $\widehat{U_3} = \widehat{-1}$ alors $-U_1$ est bien l'inverse de A modulo P. Après la boucle principale, suivant la raison de la terminaison de la boucle, on convertit la valeur $\widehat{U_3}$ ou $\widehat{U_3}$ en représentation RNS classique. Dans la suite, nous écrirons *valeurs globales* pour évoquer les valeurs $\widehat{V_1}$, $\widehat{V_1}$, $\widehat{U_3}$ et $\widehat{V_3}$.

Pour détailler la boucle principale (lignes 7–25), nous allons maintenant définir nos fonctions de base `mod4` et $\widehat{\texttt{div2r}}$.

Comme expliqué précédemment, dans le cas d'une base avec seulement des éléments impairs, la réduction modulo 4 n'est pas facile, à la différence d'une représentation binaire de position. Pour ce faire, on va repartir du théorème chinois des restes $X = \sum_{i=1}^{n} \xi_{a,i} \frac{M_a}{m_{a,i}} - qM_a$ toujours avec $\xi_{a,i} = \left| x_{a,i} \left(\frac{M_a}{m_{a,i}} \right)^{-1} \right|_{m_{a,i}}$ et $q = \left\lfloor \frac{\sum_{i=1}^{n} \xi_{a,i} \frac{M_a}{m_{a,i}}}{M_a} \right\rfloor$. Si nous prenons modulo 4 la formule du CRT, on obtient :

$$|X|_4 = \left| \sum_{i=1}^{n} |\xi_{a,i}|_4 \cdot \left| \frac{M_a}{m_{a,i}} \right|_4 - |q \cdot M_a|_4 \right|_4 . \tag{2.2}$$

Pour faciliter ce calcul, on va choisir une base $\mathcal{B}_a$ telle que $|m_{a,i}|_4 = 1$ pour tous les moduli. En fait, en dehors de l'unique modulo pair s'il y en a un, les éléments de la base valent 1 ou 3 modulo 4. Dans les implantations cryptographiques de l'état de l'art, deux bases sont toujours choisies pour pouvoir effectuer les réductions modulaires. On a donc déjà en moyenne n moduli tels que $|m_{a,i}|_4 = 1$, cette contrainte n'en est donc pas vraiment une, ou alors très faible. L'équation 2.2 devient :

$$|X|_4 = \left| \sum_{i=1}^{n} |\xi_{a,i}|_4 - |q|_4 \right|_4 . \tag{2.3}$$

La fonction `mod4` va calculer l'équation 2.3, en évaluant les deux termes $|\sum_{i=1}^{n} \xi_{a,i}|_4$ et $|q|_4$ puis en les soustrayant modulo 4. Pour calculer le premier terme, les n `EMM` $\xi_{a,i} = \left| x_{a,i} \left(\frac{M_a}{m_{a,i}} \right)^{-1} \right|_{m_{a,i}}$ doivent être effectuées. En réalité, elles peuvent être effectuées une seule fois, au début de l'algorithme. Calculer ces n produits pour une valeur $\overrightarrow{X}$ est en fait le calcul du produit RNS $\overrightarrow{X} \times \overrightarrow{T_a^{-1}}$. Il se trouve qu'une fois cette multiplication faite pour toutes les valeurs globales, toutes les autres opérations faites sont des additions de valeurs globales (ou de pré-calculs) et des divisions par 2 et 4. Le facteur $\overrightarrow{T_a^{-1}}$ est donc bien conservé tout au long de l'algorithme (on peut remarquer que c'est la première partie de la représentation $\widehat{X}$). Finalement, le calcul de ce premier terme se résume à une somme des n composantes de $\widehat{X}$ modulo 4.

Le calcul du second terme q se base, pour des raisons d'efficacité matérielle, sur l'approximation proposée par Kawamura *et al.* dans [64] et présentée en section 1.3.2. On

calcule donc :

$$q' = \left\lfloor \sigma_0 + \sum_{i=1}^{n} \frac{\texttt{trunc}(\xi_{a,i})}{2^w} \right\rfloor, \tag{2.4}$$

toujours avec `trunc` qui approche son opérande par ses t bits de poids forts (les autres étant mis à 0). Comme expliqué précédemment, t dépend des bases et de P, mais dans nos implantations FPGA et tailles de corps, t sera fixé à 6. Nous n'allons pas détailler au cas par cas pour chacune des bases la raison qui fait que c'est une valeur suffisante, mais donner un argumentaire qui fonctionne pour toutes nos bases. Pour appliquer le théorème 5 présenté à la section 1.3.2, on doit avoir $\varepsilon_{max} < \sigma_0 < 1$. Généralement dans l'état de l'art, on a $\sigma_0 = 0.5$ (voir [52, 64]). On rappelle que ε_{max} est l'erreur maximale que peut causer l'utilisation de l'approximation de Kawamura *et al.* [64], définie par $\varepsilon_{max} = n(\tau + \delta)$. On rappelle aussi que τ représente l'erreur due à l'utilisation de la fonction `trunc` et δ représente l'erreur faite en approximant les moduli par 2^w. Dans nos implantations, $(\tau+\delta)$ est proche de $\frac{1}{2^t}$ car les moduli ont été choisis très proches de 2^w. Ainsi, pour $t = 6$ on a $(\tau+\delta) \approx \frac{1}{64}$. Le plus grand n implanté est $n = 22$ (voir la table 5.5), donc $\varepsilon_{max} \approx \frac{22}{64} < 0.5$, donc on peut appliquer le théorème 5 de Kawamura *et al.* même pour notre plus grand n.

On peut maintenant remarquer que les valeurs $\xi_{a,i}$ ont déjà été calculées grâce à la représentation $\widehat{X}$. De plus, les auteurs de [64] ont prouvé que $q' = q$ lorsque $0 \leqslant n \cdot err_{max} \leqslant \sigma_0 < 1$ et $0 \leqslant X \leqslant (1 - \sigma_0)M_a$ avec err_{max} désignant l'erreur d'approximation maximale. Autrement dit, il faut que le produit M_a des éléments de la base soit suffisamment grand devant X, l'élément pour lequel on calcule le CRT, pour « contenir » l'erreur d'approximation. Il faut donc d'abord borner les valeurs qui sont traitées (classiquement par P où un petit multiple de P pour les besoins de la réduction modulaire ou les calculs sur les courbes), puis définir une base pour que le théorème soit satisfait. Dans le résultat [52] de l'état de l'art de l'implantation ECC en RNS, les paramètres choisis sont $M_a > 45P$, $0 \leqslant X < 3P$ et $\sigma_0 = 0.5$. Nous nous placerons dans ce cas pour nos implantations. Pour conclure, le calcul de q coûte n additions de t bits pour évaluer l'équation 2.4. Nous avons alors au total n additions de t bits et n additions de mots de 2 bits pour évaluer l'intégralité de l'équation 2.3.

À la différence de l'algorithme d'Euclide étendu binaire, les valeurs globales peuvent être négatives à cause des soustractions lignes 18 et 19 de l'algorithme 12, dues à l'astuce plus-minus. Si $S = V_3 - U_3$ est négatif par exemple, alors puisque le CRT est pris modulo M_a, la représentation de S en RNS sera en réalité $\overrightarrow{M_a - S}$. Si par exemple $S = -1$ alors sa représentation est $\overrightarrow{M_a - 1}$, on obtient donc un vecteur représentant un nombre arbitrairement proche de M_a. Or il nous faut garantir $X \leqslant (1 - \sigma_0)M_a$ pour garantir le résultat de l'approximation de Kawamura *et al.* équation 2.4. Nous avons donc introduit une constante additive dans la représentation $\widehat{X}$ afin de résoudre ce problème. Toutes les valeurs globales de notre algorithme `PM-MI` sont assurées d'être supérieures à $-P$. L'idée est donc de choisir une constante C_0 telle que $|C_0|_4 = 0$ et $C_0 \geqslant P$, ainsi on a $X + C_0 > 0$ et $|X + C_0| = |X|_4$. On peut par exemple choisir $C_0 = 4P$, ainsi nos conditions sont réunies et le résultat est toujours juste modulo P. Plus exactement, si l'algorithme retourne $X + C_0$, on a le bon résultat modulo P, mais dans l'intervalle $[3P, 5P]$ au lieu de $[0, 2P]$ (nous avons quand même effectué la correction à la fin de notre algorithme 12).

Nous avons maintenant toutes les briques pour construire notre représentation :

$$\widehat{X} = (\overrightarrow{X} + \overrightarrow{C_0}) \times \overrightarrow{T_a^{-1}} = \overrightarrow{X} \times \overrightarrow{T_a^{-1}} + \overrightarrow{C},$$

où $\overrightarrow{C} = \overrightarrow{C_0} \times \overrightarrow{T_a^{-1}}$. Cette deuxième version de l'équation sera utilisée pour simplifier certaines explications futures. La représentation $\widehat{X}$ permet finalement, d'une part, d'économiser les multiplications par $\overrightarrow{T_a^{-1}}$ tout au long de l'algorithme et, d'autre part, de toujours être dans l'intervalle adéquat pour effectuer l'approximation de q et être efficace en matériel.

Maintenant que nous avons réglé la question du calcul de `mod4`, il nous faut être capable de faire les divisions par 2 ou par 4 modulo P, tout en conservant la représentation $\widehat{X}$ tout au long de l'algorithme. La définition de $\widehat{\texttt{div2r}}$ dépend de la valeur $|P|_4$. Nous allons considérer par exemple que $|P|_4 = 3$. La fonction $\widehat{\texttt{div2r}}$ a 3 entrées, $\widehat{X}$, r et b_X. La donnée à diviser est $\widehat{X}$, r permet de choisir entre une division par 2 ou par 4 et enfin b_X est la valeur modulo 4 de X. En fait, l'entrée r n'est utile que pour les applications de la fonction à $\widehat{V_1}$ ou $\widehat{V_1} \pm \widehat{U_1}$. En effet, pour V_3 par exemple, on divise par 2 (resp. par 4) exclusivement lorsqu'on est divisible par 2 (resp. 4), r pourrait donc être déduit directement de b_{V_3}. Cependant, les opérations sur V_1, elles, dépendent aussi bien de r (et donc de b_{V_3}) que de b_{V_1}. Supposons que $r = 1$, alors :

$$\widehat{\texttt{div2r}}(\widehat{X}, 1, b_X) = \frac{\widehat{X}}{2} + |b_x|_2 \cdot \frac{\overrightarrow{PT_a^{-1}}}{2} + \frac{\overrightarrow{C}}{2} \tag{2.5}$$

permet d'effectuer la division par 2 tout en conservant la représentation $\widehat{X}$. Si X est impair, le deuxième terme de l'équation 2.5, c'est à dire $\frac{\overrightarrow{PT_a^{-1}}}{2}$, corrige le premier terme pour obtenir $(X + P)/2$.

La division par 4 modulo P est un peu plus complexe mais reprend exactement le même modèle. Maintenant suit la définition générale complète de $\widehat{\texttt{div2r}}$:

$$\widehat{\texttt{div2r}}(\widehat{X}, r, b_X) = \frac{\widehat{X}}{2r} + f(|b_X|_{2r}) \cdot \frac{\overrightarrow{PT_a^{-1}}}{2r} + \frac{\overrightarrow{(2r-1)C}}{2r} \quad , \tag{2.6}$$

où f permet de sélectionner ce qu'il faut rajouter comme multiple de $\frac{\overrightarrow{PT_a^{-1}}}{2r}$ pour obtenir le bon résultat modulo P. Si $|b_X|_{2r} \in \{0, 1, 2\}$, alors $f(|b_X|_{2r}) = |b_X|_{2r}$, si $|b_X|_{2r} = 3$ alors $f(|b_X|_{2r}) = -1$. Ainsi on a toujours $X + f(|b_X|_{2r}) \cdot P \equiv X \bmod P$ d'une part, et $|X + f(|b_X|_{2r}) \cdot P|_{2r} = 0$ d'autre part, ce qui nous permet de faire la division exacte par $2r$. Si on a $|P|_4 = 1$, il suffit d'échanger les valeurs de $f(1)$ et $f(3)$. On remarque qu'on retrouve bien l'équation 2.5 si on pose $r = 1$. Pour vérifier que la représentation $\widehat{X}$ est bien conservée, on obtient en développant :

$$\begin{aligned}
\widehat{\texttt{div2r}}(\widehat{X}, r, b_X) &= \frac{\widehat{X}}{2r} + f(|b_X|_{2r}) \cdot \frac{\overrightarrow{PT_a^{-1}}}{2r} + \frac{\overrightarrow{(2r-1)C}}{2r} \\
&= \frac{\overrightarrow{XT_a^{-1} + C}}{2r} + f(|b_X|_{2r}) \cdot \frac{\overrightarrow{PT_a^{-1}}}{2r} + \frac{\overrightarrow{(2r-1)C}}{2r} \\
&= \frac{\overrightarrow{(X + f(|b_X|_{2r})P)T_a^{-1}}}{2r} + \overrightarrow{C} \\
&= \widehat{\left|\frac{X}{2r}\right|_P} \quad .
\end{aligned}$$

Le coût de $\widehat{\texttt{div2r}}$ est d'une multiplication par canal et 2 additions, soient n `EMM`$+2n$ `EMA`. Puisque l'on additionne à notre entrée des constantes, on peut pré-calculer ces sommes là

pour descendre à $n\,\texttt{EMM} + n\,\texttt{EMA}$ (6 sommes à pré-calculer).

Pour finir, nous allons maintenant détailler la boucle principale (lignes 7–25) qui est le cœur de notre algorithme. Tout d'abord, comme pour les autres algorithmes d'Euclide étendus binaires, on effectue une boucle où on divise $\widehat{V_3}$ tant qu'il est pair. Vu qu'on a la valeur modulo 4 de $\widehat{V_3}$, on peut directement diviser par 4 pour réduire le nombre de tours. Pour chaque division par 2 ou 4, on reporte la même opération modulo P sur la valeur $\widehat{V_1}$. On calcule ensuite les nouvelles valeurs modulo 2 ou 4 de $\widehat{V_1}$ et $\widehat{V_3}$, puis on met à jour le compteur du nombre de divisions par 2 qu'on a effectué sur $\widehat{V_3}$, c'est à dire l_v. Après cette boucle, on stocke les valeurs de $\widehat{V_3}$ et $\widehat{V_1}$ en prévision d'un échange avec les valeurs $\widehat{U_3}$ et $\widehat{U_1}$. Vient ensuite le test plus-minus, qui n'appelle pas la fonction `mod4`, il suffit de traiter avec les valeurs b_{V_3} et b_{U_3} directement. On divise la somme ou la différence en conséquence, en corrigeant les entrées pour qu'elles aient le bon format pour $\widehat{\texttt{div2r}}$. Par exemple, lorsque l'on calcule $\widehat{V_3} + \widehat{V_1}$, le terme additif C est compté 2 fois : on soustrait donc le terme en trop. Ces corrections peuvent aussi être incluses dans les pré-calculs, nécessitant 8 valeurs supplémentaires. On compare finalement l_v et l_u pour équilibrer le nombre de divisions entre les 2 opérandes. On rappelle que ce sont des petits nombres : $l_u, l_v < \log_2 P$.

Remarque. Le but de l'utilisation du plus-minus est initialement d'éviter les comparaisons difficiles. On les évite en réalisant des modulo 4 efficacement. Il se trouve que lorsqu'on possède une telle fonction, on pourrait penser qu'on est forcément capable de comparer 2 nombres (et d'utiliser l'algorithme d'Euclide étendu binaire en RNS). Il est vrai que si $X < Y$, alors le CRT effectué sur $\overrightarrow{X - Y}$ retourne l'entier $X - Y + M_a$. Puisque tous les moduli $m_{a,i}$ sont impairs, M_a est impair donc $|X - Y + M_a|_2 \neq |X - Y|_2$. On rappelle que $|X - Y|_2$ peut être obtenu à partir de b_X et b_Y. Inversement, si $X > Y$, $|X - Y + M_a|_2 = |X - Y|_2$, ce qui nous donne un test pour comparer X et Y. Le problème est que notre fonction `mod4` utilise une approximation (celle de [64]), qui ne garantit plus un résultat juste quand on calcule $|X - Y + M_a|_2$, on ne peut donc pas faire de comparaison avec notre proposition en utilisant [64]. Si par contre on est capable d'évaluer de façon exacte le modulo en utilisant une conversion vers la représentation MRS (en utilisant [120]), alors on a plus besoin de l'astuce plus-minus, mais la fonction `mod4` coûterait alors bien plus cher que celle proposée car la conversion vers le MRS coûte $\frac{n(n-1)}{2}$ `EMM`, et est en plus difficilement parallélisable et donc non adaptée à l'architecture `Cox-Rower` (voir section 1.3.2).

Pour conclure cette section, on peut remarquer que tout l'algorithme s'effectue sur une seule base au lieu de 2 pour l'algorithme `FLT-MI`. Ceci est dû au fait que l'on n'a pas besoin d'extension de base. En fait, il y a quand même une évaluation du CRT, mais réduite modulo 4, avec un coût très faible par rapport à une véritable extension. Effectuer ce petit calcul dans le `Cox` revient à extraire l'information minimale pour faire l'algorithme, là où une extension dans un modulo pair (comme les premières stratégies proposées) complique les calculs sans accélérer l'algorithme. Dans la section 2.4 sera fournie une évaluation du coût à partir de statistiques sur le nombre de tours de boucle.

2.3 Algorithme binaire-ternaire plus-minus en RNS

L'algorithme binaire d'Euclide est basé sur la facilité des divisions par 2 et des réductions modulo 2 dans la représentation binaire. Bien que nous ayons adapté cet algorithme pour le RNS de façon efficace dans la section précédente, cette section pose la question de la possibilité d'étendre un tel algorithme vers d'autres tests de divisibilité. Un nou-

vel algorithme d'inversion modulaire, appelé algorithme binaire-ternaire plus-minus RNS (`BTPM-MI`), et qui utilise des réductions modulo 3, 6 et 12 en plus de celles par 2 et 4 de la section précédente, est proposé. L'algorithme complet est présenté algorithme 18. Il n'a pas encore été implanté sur FPGA, ce qui est une première perspective de travail.

Avant de présenter le nouvel algorithme, nous allons nous demander ce que sont les divisons par 2 et 4 en RNS. Ce ne sont pas simplement des décalages à la manière de la représentation binaire standard. Elles peuvent être effectuées de 2 façons différentes. La première solution consiste à multiplier par $2^{-1} \bmod m_{a,i}$ et $4^{-1} \bmod m_{a,i}$ pour chacun des $m_{a,i}$, afin d'utiliser une architecture similaire à celle de l'état de l'art [52]. En effet, cette architecture peut très bien effectuer des multiplications par des constantes, comme requis pour les calculs dans ECC, mais ne possède rien de particulier pour des divisions par 2. Dans une telle architecture, diviser par 2, 6, 12, 2 ou 4 est donc aussi dur : il faut juste multiplier par l'inverse du diviseur dans chacun des $m_{a,i}$. Cela induit quand même d'avoir pré-calculé les différents inverses, et donc de les stocker.

Une autre solution peut être utilisée, basée sur des décalages. En effet, même si un décalage sur toute la représentation RNS n'a pas de sens (ou en tout cas pas celui que l'on recherche), on peut comprendre ce qu'il se passe au niveau de chacun des moduli. Du fait que les valeurs sur chacun des canaux soient en représentation binaire classique, on sait ce qu'est un décalage. Par contre, les calculs sur les canaux sont des calculs modulaires, c'est pour ça qu'on ne peut pas simplement décaler sur toute la longueur du vecteur RNS. Par exemple, soit $m_{a,i}$ impair et $x_{a,i}$ une valeur du canal que l'on souhaite diviser par 2. Si $x_{a,i}$ est pair, un décalage suffit pour effectuer la division. Si $x_{a,i}$ est impair, l'opération à effectuer est $s = \frac{x_{a,i}+m_{a,i}}{2}$ car dans ce cas $s \equiv x_{a,i}/2 \bmod m_{a,i}$ et on a bien $0 < s < m_{a,i}$ ($x_{a,i} < m_{a,i}$). Les désavantages de cette méthode sont que, dans notre architecture actuelle, basée sur celle de l'état de l'art [52], une addition coûte autant qu'une multiplication (voire plus cher en temps). En effet, l'architecture est optimisée pour accélérer la réduction modulaire (ce qui est normal au vu du coût de celle-ci), les `Rower`s contiennent donc des multiplieurs-accumulateurs pour calculer rapidement l'extension de base. Une addition requiert donc 2 cycles (on calcule $a \times 1 + b \times 1$ en réalité). Utiliser une multiplication est donc plus avantageux sur cette architecture. De plus, elle implique un contrôle interne à chacun des moduli, car les $x_{a,i}$ sur les différents canaux sont indépendamment pairs ou impairs. Ainsi, cette division par 2 impliquerait une addition uniquement sur une partie des moduli, ce qui complique par rapport au contrôle habituel où les calculs sur tous les `Rower`s sont synchronisés et de même nature.

Pour une future implantation FPGA, les deux solutions doivent être évaluées en matériel. Bien qu'il soit immédiat d'utiliser la multiplication par l'inverse (et plus simple en terme de contrôle), il est difficile *a priori* de comparer les 2 approches. Si l'implantation de la division par 2 ou 4 avec décalage se révèle bien plus efficace, cela réduirait alors l'intérêt alors de l'utilisation de divisions par 3 (sauf si une technique peu coûteuse peut être utilisée pour diviser par 3). À partir de maintenant, pour l'évaluation de l'algorithme et les comparaisons, nous allons considérer que les divisions exactes par de petites constantes sont faites uniquement en multipliant par l'inverse pré-calculé, c'est à dire qu'une division par 2 ou par 4 vaut autant qu'une division par 3 ou 6.

Avant de détailler l'algorithme 18, nous allons évaluer l'autre opération critique qu'est la réduction modulaire par 3. En effet, la réduction modulo 4 a déjà été traitée dans la section

Algorithme 18: Inversion modulaire binaire-ternaire plus-minus proposée (`BTPM-MI`).

Entrées : $\overrightarrow{A}, P > 2$ avec $\gcd(A, P) = 1$
Pré-calculs : $\overrightarrow{T_a}, \overrightarrow{T_a^{-1}}$, $|P|_4$ et l'ensemble de pré-calculs pour `Divup`
Sortie : $\overrightarrow{S} = \overrightarrow{|A^{-1}|_P}$, $0 \leqslant S < 2P$

$l_u \leftarrow 0$, $l_v \leftarrow 0$, $\widehat{U_1} \leftarrow \widehat{0}$, $\widehat{V_1} \leftarrow \widehat{1}$, $\widehat{V_3} \leftarrow \widehat{A}$
$b_{U_1} \leftarrow 0$, $t_{U_1} \leftarrow 0$, $b_{U_3} \leftarrow |P|_4$, $t_{U_3} \leftarrow |P|_3$
$b_{V_1} \leftarrow 1$, $t_{V_1} \leftarrow 1$, $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $t_{V_3} \leftarrow \texttt{mod3}(\widehat{V_3})$
tant que $\widehat{V_3} \neq \widehat{1}$ *et* $\widehat{U_3} \neq \widehat{1}$ *et* $\widehat{V_3} \neq \widehat{-1}$ *et* $\widehat{U_3} \neq \widehat{-1}$ **faire**
  tant que $|b_{V_3}|_2 = 0$ *ou* $t_{V_3} = 0$ **faire**
    $\texttt{Divup}\left(\widehat{V_3}, b_{V_3}, t_{V_3}, \widehat{V_1}, b_{V_1}, t_{V_1}, v\right)$
  $\widehat{V_3^*} \leftarrow \widehat{V_3}$, $\widehat{V_1^*} \leftarrow \widehat{V_1}$
  si $|t_{V_3} + t_{U_3}|_3 = 0$ **alors**
    si $|b_{V_3} + b_{U_3}|_4 = 0$ **alors**
      $r = 1$
      $\widehat{V_3} \leftarrow \widehat{\texttt{div12}}(\widehat{V_3} + \widehat{U_3} - \overrightarrow{C}, 0, 0)$
      $\widehat{V_1} \leftarrow \widehat{\texttt{div12}}(\widehat{V_1} + \widehat{U_1} - \overrightarrow{C}, |b_{V_1} + b_{U_1}|_4, |t_{V_1} + t_{U_1}|_3)$
    sinon
      $r = 0$
      $\widehat{V_3} \leftarrow \widehat{\texttt{div6}}(\widehat{V_3} + \widehat{U_3} - \overrightarrow{C}, 2, 0)$
      $\widehat{V_1} \leftarrow \widehat{\texttt{div6}}(\widehat{V_1} + \widehat{U_1} - \overrightarrow{C}, |b_{V_1} + b_{U_1}|_4, |t_{V_1} + t_{U_1}|_3)$
  sinon
    si $|b_{V_3} - b_{U_3}|_4 = 0$ **alors**
      $r = 1$
      $\widehat{V_3} \leftarrow \widehat{\texttt{div12}}(\widehat{V_3} - \widehat{U_3} + \overrightarrow{C}, 0, 0)$
      $\widehat{V_1} \leftarrow \widehat{\texttt{div12}}(\widehat{V_1} - \widehat{U_1} + \overrightarrow{C}, |b_{V_1} - b_{U_1}|_4, |t_{V_1} - t_{U_1}|_3)$
    sinon
      $r = 0$
      $\widehat{V_3} \leftarrow \widehat{\texttt{div6}}(\widehat{V_3} - \widehat{U_3} + \overrightarrow{C}, 2, 0)$
      $\widehat{V_1} \leftarrow \widehat{\texttt{div6}}(\widehat{V_1} - \widehat{U_1} + \overrightarrow{C}, |b_{V_1} - b_{U_1}|_4, |t_{V_1} - t_{U_1}|_3)$
  si $l_v > l_u$ **alors**
    $\widehat{U_3} \leftarrow \widehat{V_3^*}$, $b_{U_3} \leftarrow b_{V_3}$, $t_{U_3} \leftarrow t_{V_3}$
    $\widehat{U_1} \leftarrow \widehat{V_1^*}$, $b_{U_1} \leftarrow b_{V_1}$, $t_{U_1} \leftarrow t_{V_1}$
    $\text{swap}(l_u, l_v)$
  $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $t_{V_3} \leftarrow \texttt{mod3}(\widehat{V_3})$
  $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$, $t_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$
  $l_v \leftarrow l_v + r + \gamma$
si $\widehat{V_3} = \widehat{1}$ **alors retourner** $(\widehat{V_1} - \overrightarrow{C})\overrightarrow{T_a} + \overrightarrow{P}$
sinon si $\widehat{U_3} = \widehat{1}$ **alors retourner** $(\widehat{U_1} - \overrightarrow{C})\overrightarrow{T_a} + \overrightarrow{P}$
sinon si $\widehat{V_3} = \widehat{-1}$ **alors retourner** $-(\widehat{V_1} - \overrightarrow{C})\overrightarrow{T_a} + \overrightarrow{P}$
sinon retourner $-(\widehat{U_1} - \overrightarrow{C})\overrightarrow{T_a} + \overrightarrow{P}$

précédente, la combinaison avec une réduction modulo 3 nous permet automatiquement d'obtenir les valeurs modulo 6 et 12. En repartant encore de la formule du CRT que l'on prend modulo 3, on obtient :

$$|X|_3 = \left| \sum_{i=1}^{n} \xi_{a,i} - q \right|_3 \quad , \tag{2.7}$$

en choisissant $\mathcal{B}_a$ tel que $|m_{a,i}|_3 = 1$ pour tout i. Dans cette équation, une seule réduction modulo 3 est effectuée après la somme (à la différence de l'équation 2.3 où les réductions sont faites au fur et à mesure). En représentation binaire de position, la réduction modulo 3 de $\xi_{a,i}$ va dépendre de tous les bits de $\xi_{a,i}$ à la différence du modulo 4 qui ne dépend que des 2 derniers. Ainsi, effectuer la somme des $\xi_{a,i}$ avant de réduire semble le meilleur choix, une future évaluation FPGA devra être effectuée. Le coût d'une réduction modulaire est donc de n `EMA` et 1 réduction modulo 3 de $w + \log_2 n$ bits. Le calcul de q est déjà effectué par le modulo 4 lorsque modulo 4 et modulo 3 sont effectués conjointement. Le calcul efficace de valeurs modulo 3, et plus généralement modulo un petit nombre premier sera étudié dans le chapitre 5.

Nous allons maintenant étudier la structure globale de l'algorithme 18 et détailler les sous-fonctions qui le composent. La structure est très similaire à celle du `PM-MI`. On retrouve en premier lieu la boucle sur la parité de $\widehat{V_3}$, qui peut maintenant être effectuée aussi lorsque $\widehat{V_3}$ est divisible par 3 (t_X représente la valeur de X modulo 3 dans l'algorithme). Par soucis de clarté, les calculs de cette sous-boucle sont encapsulés dans la fonction `Divup` (détaillée dans la suite). Ensuite vient une nouvelle forme du plus-minus. Il se trouve que si X et Y ne sont ni multiples de 2, ni de 3 alors $X + Y$ ou $X - Y$ est un multiple de 6. Il suffit de remarquer que les seules valeurs possibles pour X et Y sont 1 et 5 modulo 6. On sait déjà que $X + Y$ et $X - Y$ sont tous deux pairs, il ne nous reste plus qu'à procéder à un test de la somme modulo 3 pour effectuer le nouveau plus-minus. Aux lignes 9 et 18 de l'algorithme 18, on teste si le chemin choisi par le plus-minus n'est pas seulement divisible par 6 mais aussi par 12. Puisque nous avons déjà les valeurs de U_3 et V_3 modulo 4, ce test en plus ne coûte rien. Par contre, la division par 12 engendrée demandera des pré-calculs supplémentaires (bien qu'une bonne partie des pré-calculs nécessaires à la division par 12 modulo P sont déjà pré-calculés pour les divisions par 4 et 6). Enfin, une dernière différence avec le `PM-MI` se trouve dans le contrôle de la taille des valeurs. Pour cette version binaire-ternaire, l_u et l_v comptabilisent le nombre de divisions par 2 et par 3 qui ont été effectuées sur $\widehat{U_3}$ et $\widehat{V_3}$. Ces compteurs permettent en fait d'approcher la taille de nos valeurs globales tout au long de l'algorithme, c'est d'autant plus vrai si A a un nombre de bits proche de $\log_2 P$, ce qui est le cas en pratique pour un élément de $\mathbb{F}_P$ pris au hasard. Pour comptabiliser une division par 3 on va donc ajouter $\log_2 3$ à l_u ou l_v, ou plutôt une valeur approchée γ. Pour l'évaluation de notre algorithme section 2.4, $\gamma = 1.5$ permet d'obtenir de bons résultats. En pratique, l'augmentation de la précision de γ est négligeable.

La fonction `Divup` est détaillée dans l'algorithme 19. Cette fonction est une adaptation des lignes 10, 11 et 12 de l'algorithme 12. On se retrouve maintenant avec 5 cas à traiter car $\widehat{V_3}$ peut être divisé par 2, 3, 4, 6 ou 12. La fonction change directement la valeur de ses entrées : elle divise et effectue une mise à jour des valeurs nécessaires. Les différentes fonctions $\widehat{\texttt{div2}}$ à $\widehat{\texttt{div12}}$ sont construites exactement sur le modèle de $\widehat{\texttt{div2r}}$, elles retournent les résultats des divisions modulo P dans la représentation $\widehat{X}$. D'ailleurs $\widehat{\texttt{div2}}$ et $\widehat{\texttt{div4}}$ sont les deux sous-cas de la fonction $\widehat{\texttt{div2r}}$ définie pour le `PM-MI`. Les fonctions `mod4` et `mod3` ont

Algorithme 19: Fonction `Divup`.

Entrées : $\widehat{V_3}$, b_{V_3}, t_{V_3}, $\widehat{V_1}$, b_{V_1}, t_{V_1},v

Pré-calculs : $\overrightarrow{C/2}$, $\overrightarrow{(2\,C/3)}$, $\overrightarrow{(3\,C/4)}$, $\overrightarrow{(5\,C/6)}$, $\overrightarrow{(11\,C/12)}$, $\overrightarrow{(PT_a^{-1})/12}$, $\overrightarrow{(PT_a^{-1})/6}$, $\overrightarrow{(PT_a^{-1})/4}$, $\overrightarrow{(PT_a^{-1})/3}$ $\overrightarrow{(5PT_a^{-1})/12}$, $\overrightarrow{(PT_a^{-1})/2}$, $\overrightarrow{(-PT_a^{-1})/12}$, $\overrightarrow{(-PT_a^{-1})/6}$ $\overrightarrow{(-PT_a^{-1})/4}$, $\overrightarrow{(-PT_a^{-1})/3}$, $\overrightarrow{(-5PT_a^{-1})/12}$

Sorties : $\widehat{V_3}$, b_{V_3}, t_{V_3}, $\widehat{V_1}$, b_{V_1}, t_{V_1}, l_v

cas où $b_{V_3} = 0$ *et* $t_{V_3} = 0$
- $\widehat{V_3} \leftarrow \widehat{\texttt{div12}}\left(\widehat{V_3}, b_{V_3}, t_{V_3}\right)$
- $\widehat{V_1} \leftarrow \widehat{\texttt{div12}}\left(\widehat{V_1}, b_{V_1}, t_{V_1}\right)$
- $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$
- $t_{V_3} \leftarrow \texttt{mod3}(\widehat{V_3})$, $t_{V_1} \leftarrow \texttt{mod3}(\widehat{V_1})$
- $v \leftarrow v + \gamma + 2$

cas où $b_{V_3} = 2$ *et* $t_{V_3} = 0$
- $\widehat{V_3} \leftarrow \widehat{\texttt{div6}}\left(\widehat{V_3}, b_{V_3}, t_{V_3}\right)$
- $\widehat{V_1} \leftarrow \widehat{\texttt{div6}}\left(\widehat{V_1}, b_{V_1}, t_{V_1}\right)$
- $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$
- $t_{V_3} \leftarrow \texttt{mod3}(\widehat{V_3})$, $t_{V_1} \leftarrow \texttt{mod3}(\widehat{V_1})$
- $v \leftarrow v + \gamma + 1$

cas où $b_{V_3} = 0$ *et* $t_{V_3} \neq 0$
- $\widehat{V_3} \leftarrow \widehat{\texttt{div4}}\left(\widehat{V_3}, b_{V_3}\right)$,
- $\widehat{V_1} \leftarrow \widehat{\texttt{div4}}\left(\widehat{V_1}, b_{V_1}\right)$,
- $t_{V_3} \leftarrow \texttt{update3}(t_{V_3}, b_{V_3}, b_{V_3})$, $t_{V_1} \leftarrow \texttt{update3}(t_{V_1}, b_{V_1}, b_{V_3})$
- $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$
- $v \leftarrow v + 2$

cas où $b_{V_3} \neq 0, 2$ *et* $t_{V_3} = 0$
- $\widehat{V_3} \leftarrow \widehat{\texttt{div3}}\left(\widehat{V_3}, t_{V_3}\right)$
- $\widehat{V_1} \leftarrow \widehat{\texttt{div3}}\left(\widehat{V_1}, t_{V_1}\right)$,
- $b_{V_3} \leftarrow \texttt{update4}(b_{V_3}, t_{V_3})$, $b_{V_1} \leftarrow \texttt{update4}(b_{V_1}, t_{V_1})$
- $t_{V_3} \leftarrow \texttt{mod3}(\widehat{V_3})$, $t_{V_1} \leftarrow \texttt{mod3}(\widehat{V_1})$
- $v \leftarrow v + \gamma$

cas où $b_{V_3} = 2$ *et* $t_{V_3} \neq 0$
- $\widehat{V_3} \leftarrow \widehat{\texttt{div2}}\left(\widehat{V_3}, b_{V_3}\right)$
- $\widehat{V_1} \leftarrow \widehat{\texttt{div2}}\left(\widehat{V_1}, b_{V_1}\right)$
- $t_{V_3} \leftarrow \texttt{update3}(t_{V_3}, b_{V_3}, b_{V_3})$, $t_{V_1} \leftarrow \texttt{update3}(t_{V_1}, b_{V_1}, b_{V_3})$
- $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$
- $v \leftarrow v + 1$

retourner $\widehat{V_3}$, b_{V_3}, t_{V_3}, $\widehat{V_1}$, b_{V_1}, t_{V_1}, l_v

déjà été expliquées précédemment, il ne manque plus qu'à détailler les fonctions `update3` et `update4`. Ces fonctions évaluent respectivement la nouvelle valeur modulo 3 ou modulo 4 lorsqu'on a pas besoin de faire tout le calcul `mod3` ou `mod4`. Par exemple, ligne 13 de l'algorithme 19 commence le cas ou $\widehat{V_3}$ doit être divisé par 4. Dans ce cas, nous n'avons pas besoin de `mod3` pour calculer la nouvelle valeur t_{V_3}. La fonction `update3`$(t_{V_3}, b_{V_3}, b_{V_3})$ en ligne 16 va retourner $t_{V_3} \cdot 4^{-1} \bmod 3 = t_{V_3}$ et $t_{V_3} \cdot 2^{-1} \bmod 3 = 2t_{V_3} \bmod 3$ à la ligne 28. Les opérations sur $\widehat{V_1}$ étant plus complexes que celles sur $\widehat{V_3}$, la fonction `update3` retourne un résultat de la forme $(t_{V_1} + cP) \cdot 4^{-1} \bmod 3$. En effet, dans le cas de $\widehat{V_1}$ on doit effectuer une division modulo P et non pas une simple division exacte. Le c est celui qui est calculé pour effectuer la division avec $\widehat{\texttt{div4}}$, par exemple si $b_{V_1} = 2$, alors on calcule $(t_{V_1} + 2P) \cdot 4^{-1} \bmod 3$ (la valeur de c est déterminée dans l'équation 2.6). Finalement, lorsqu'on calcule `update3`$(t_{V_1}, b_{V_1}, b_{V_3})$, la première entrée correspond à ce qu'on doit modifier, la seconde permet de déterminer le c qu'on doit ajouter et la troisième quelle division nous avons effectué sur $\widehat{V_1}$. La fonction `update4` effectue la même chose mais pour la mise à jour des valeurs modulo 4 lorsqu'on divise par 3. Il n'y a plus que 2 entrées pour `update4` car elle est utilisée que dans un seul cas (ligne 19 de l'algorithme 19).

Avec `Divup` défini, l'algorithme est maintenant complètement décrit. Pour résumer, l'algorithme utilise le fait que les divisions par 2 et par 3 sont aussi difficiles et que la réduction par 3 n'est pas très difficile. Ainsi, grâce à ces plus grandes divisions on va avancer plus vite dans l'algorithme et réduire le nombre d'itérations de la boucle principale. En fait, comme on le verra dans la partie comparaison, le nombre d'itérations de la boucle interne va augmenter un petit peu mais ce surcoût sera largement compensé par la réduction du nombre d'itérations de la boucle principale. On pourrait avoir dans l'idée d'utiliser d'autres diviseurs, plus grands (par exemple 5 ou 7) afin de réduire encore le nombre d'itérations. En effet, la division ne coûte pas plus cher et la réduction modulaire par 5 ou 7 serait même presque gratuite : en effet dans l'équation 2.7 de réduction modulo 3 on effectue la somme puis on réduit par 3 les $w + \log_2 n$ bits obtenus. Cette somme est en fait commune aux réductions modulo 3, 5 et 7, le surcoût serait juste une réduction modulo 5 et une modulo 7 de ce petit nombre. Nous verrons dans le chapitre 5 comment factoriser ces réductions modulaires dans la numération simple de position pour d'autres applications. Le problème de cette approche pour l'inversion est qu'elle rajoute, en contrepartie, beaucoup de pré-calculs et beaucoup de cas pour la fonction `Divup`. La liste des pré-calculs est déjà bien rallongée entre ceux de l'algorithme 12 et ceux (cumulés) des algorithmes 18 et 19. Rajouter de plus grandes valeurs augmenterait grandement le nombre de pré-calculs. De plus, le gain en terme du nombre d'opérations s'amenuise : une valeur globale a une probabilité $\frac{1}{5}$ d'être multiple de 5, $\frac{1}{7}$ d'être multiple de 7 et ainsi de suite. Nous pensons donc qu'il est très probablement préférable de s'arrêter au modulo 3.

2.4 Comparaison avec l'état de l'art

2.4.1 Complexité du `FLT-MI`

L'algorithme de l'état de l'art (`FLT-MI`) ainsi que les deux algorithmes proposés dans nos travaux (`PM-MI` et `BTPM-MI`) effectuent $O(\ell)$ itérations de la boucle principale pour des opérandes en entrée de ℓ bits. La principale différence vient donc du coût des calculs internes à cette boucle. Pour l'évaluation de complexité on introduit certaines notations pour les opérations élémentaires :

– cox-add est une addition de deux éléments de t bits (effectuée dans le Cox) ;

– mod4-add est une addition de deux éléments de 2 bits modulo 4 (ces opérations sont comptées mais leur coût est très faible comparé aux `EMA` et `EMM`) ;
– mod3 est une réduction modulo 3 sur $w + \log_2 n$ bits.

Dans le but d'évaluer le coût de la version `FLT-MI` présentée dans l'algorithme 16, on doit évaluer le coût de chaque itération de la boucle principale. En moyenne, `MM` est exécutée avec une probabilité 1/2 à la ligne 5 et celle présente à la ligne 6 est toujours effectuée. Chaque itération calcule donc 1.5 `MM`. Nous considérons ici qu'une `MM` coûte $2n^2 + 6n$ `EMM`, $2n^2 + n$ `EMA` et $2n$ cox-add, comme Gandino *et al.* [48] (voir section 1.3.3 pour l'évaluation de `MM`). Pour conclure, l'algorithme 16 a une complexité moyenne de $O(\ell \times n^2)$.

On peut remarquer qu'à travers ce chapitre, nous avons exclusivement considéré le cas de l'extension de base utilisant le CRT (notée CRTBE) avec la méthode de Kawamura *et al.* [64]. L'autre méthode principale est d'utiliser les extensions de base en passant par le MRS (notée MRSBE [120]). Comme déjà mentionné, cette méthode est moins parallélisable et donc moins bien adaptée à notre implantation matérielle. Même en mettant de côté cet aspect implantation pour se concentrer uniquement sur le nombre d'opérations, la multiplication modulaire via MRS coûte $3n^2+4n$ `EMM`, donc plus cher que la CRTBE. Nous avons présenté dans l'état de l'art des optimisations proposées par Bajard *et al.* dans [12] pour réduire ce coût, grâce à des bases bien choisies. Même si ces optimisations réduisent le coût en `EMM` et `EMA` de MRSBE, elles ne sont toujours pas suffisamment parallélisables pour les considérer dans le cadre de l'architecture RNS ECC de l'état de l'art. On pourrait effectuer une étude plus fine, mais la complexité de `FLT-MI` en `EMM` et `EMA` ne varie pas énormément en choisissant l'une ou l'autre des extensions de base. Dans les deux cas, `FLT-MI` est en $O(\ell \times n^2)$ `EMM` et `EMA`. Nous allons voir que de toute façon, les gains obtenus grâce à nos algorithmes sont tels qu'il n'est pas nécessaire de faire différentes études de cas. Pour conclure, dans un contexte matériel, il suffit de se concentrer sur CRTBE pour effectuer notre comparaison.

2.4.2 Complexité de `PM-MI` et `BTPM-MI`

Afin d'évaluer le coût de l'algorithme 17 `PM-MI`, on doit d'abord estimer le coût de `mod4` et $\widehat{\texttt{div2r}}$. Le calcul de `mod4` comprend le calcul de q, qui requiert n cox-add et la somme modulo 4 finale, qui requiert n mod4-add ($|q|_4 + \sum_{i=1}^{n} |\xi_{a,i}|_4$). Comme expliqué dans la section 2.2, $\widehat{\texttt{div2r}}$ coûte n `EMM` (la multiplication par 4^{-1} ou 2^{-1}) et n `EMA` (grâce aux pré-calculs des sommes des constantes).

Ensuite, il faut évaluer le nombre d'itérations de la boucle interne (lignes 8–12) ainsi que le coût de chacune de celles-ci. En fait, la probabilité qu'une première itération ait lieu est de $\frac{1}{2}$ (on teste si $|V_3|_2 = 0$). Une seconde itération n'arrive que si V_3 est un multiple de 8, puisque s'il était juste un multiple de 4, on aurait divisé par 4 durant la première itération. On a donc une probabilité $\frac{1}{8}$ d'avoir 2 itérations. En continuant ainsi, la probabilité de faire une j^{e} itération est de $\frac{1}{2 \cdot 4^{j-1}}$. Finalement, en moyenne on obtient $\frac{1}{2}\sum_{j=0}^{\infty} \frac{1}{4^j} = \frac{2}{3}$ itérations de la boucle interne pour une itération de la boucle principale. Dans une itération de la boucle interne, 2 `mod4` et 2 $\widehat{\texttt{div2r}}$ sont effectués, menant à $2n$ cox-add, $2n + 2$ mod4-add, $2n$ `EMM` et $2n$ `EMA`.

Après la boucle interne, la fin de la boucle principale va calculer 2 $\widehat{\texttt{div2r}}$ et 2 `mod4` (lignes 14–25). On peut voir que les entrées de $\widehat{\texttt{div2r}}$ sont de la forme $\widehat{X} \pm \widehat{Y} \pm \overrightarrow{C}$, mais ces additions sont directement intégrées dans les pré-calculs. Aucun nouveau pré-calcul

n'est nécessaire, juste un peu de contrôle sur le choix de la constante que l'on ajoute. Pour ne pas compliquer inutilement l'écriture mathématique de $\widehat{\texttt{div2r}}$, nous avons préféré cette présentation dans l'algorithme. Finalement, le coût des lignes 14–25 est de $2n$ `EMM`, $4n$ `EMA`, $2n$ cox-add et $2n + 2$ mod4-add.

Une évaluation complète du nombre de tours de boucle de cet algorithme est complexe. Nous avons effectué des tests sur plus de 700 000 valeurs avec des tailles ℓ cryptographiques (voir les standards du NIST [91]). En moyenne nous obtenons 0.71ℓ itérations de boucle principale par inversion. Dans [66, p. 348–353], ce nombre est évalué à environ 0.70597ℓ pour l'algorithme d'Euclide étendu binaire classique, on est donc très proche de ce résultat théorique. Pour conclure sur notre premier algorithme, sa complexité moyenne est de $O(\ell \times n)$ `EMM` et `EMA` contre $O(\ell \times n^2)$ pour l'état de l'art `FLT-MI`.

Pour chiffrer le coût de notre second algorithme 18, nous avons aussi effectué 700 000 tests pour avoir une moyenne du nombre de chacune des différentes opérations. Il y a moins d'itérations de boucle principale que dans le `PM-MI` simple, en moyenne 0.46ℓ au lieu de 0.71ℓ. En revanche, il y a plus d'itérations de la boucle interne, ce qui est normal puisqu'on a plus de diviseurs. En moyenne, on a observé 3/4 itérations de boucle interne pour 1 tour de boucle principale, contre 2/3 pour le `PM-MI`. Finalement le nombre d'opérations a été réduit avec le nouvel algorithme.

Les tables 2.1 et 2.2 donnent le détail du nombre d'opérations pour certaines tailles de corps et certaines bases pour les 3 algorithmes (nos 2 propositions et l'état de l'art). Le nombre de multiplications et d'additions élémentaires a été très significativement réduit par rapport au `FLT-MI` avec nos deux algorithmes. Le nombre de multiplications élémentaires `EMM` (qui est la métrique classique utilisée pour comparer les algorithmes RNS) est divisé par 12 à 37 avec le `PM-MI` et par 18 à 54 par le `BTPM-MI` comparé au `FLT-MI`, suivant les différents paramètres ℓ et n choisis. Par exemple, pour un P tiré aléatoirement de 192 bits avec $n = 12$ et $w = 17$, alors `FLT-MI` requiert 103140 `EMM` contre seulement 5474 `EMM` pour `PM-MI` et 3744 pour `BTPM-MI` en moyenne. Bien sûr, à cause de la différence de complexité en n, plus la base a d'éléments, plus le rapport entre nos algorithmes et le `FLT-MI` est grand. Par exemple, pour une taille de mots élémentaires fixée w, n va croître avec $\ell = \lceil \log_2 P \rceil$ puisque par définition, $n \times w = \ell$.

Comme présenté finalement dans la figure 2.1, `BTPM-MI` permet de réduire le nombre de `EMM` de 30 % et celui des `EMA` de 20 %, quelle que soit la taille du corps ℓ et quel que soit le nombre d'éléments dans la base n par rapport à `PM-MI`, le rapport des complexités étant constant.

2.5 Architecture et implantation FPGA

L'algorithme de l'état de l'art `FLT-MI` et notre inversion `PM-MI` ont été implantés en FPGA. La version binaire-ternaire `BTPM-MI` ne dispose pas encore d'implantation, mais les premiers éléments présentés dans la section 2.4 semblent prometteurs. Nous avons comme objectif final la conception complète d'un cryptoprocesseur RNS pour ECC. Nous avons donc implanté une architecture `Cox-Rower` très similaire à celle de Guillermin [52], qui est l'état de l'art en terme de performances pour un cryptoprocesseur ECC en RNS avec protection contre certaines attaques par canaux cachés.

Algo.	ℓ	$n \times w$	nombre d'opérations				
			`EMM`	`EMA`	cox-add	mod4-add	mod3
`FLT-MI` aléa.	192	12×17	103140	85950	6876	0	0
		9×22	61884	48991	5157	0	0
		7×29	40110	30083	4011	0	0
	256	12×22	137700	114750	9180	0	0
		9×29	82620	65407	6885	0	0
		8×33	67320	52020	6120	0	0
	384	18×22	434322	382617	20682	0	0
		14×29	273462	233247	16086	0	0
		12×33	206820	172350	13788	0	0
	521	19×29	652080	577980	29640	0	0
		16×33	474240	411840	24960	0	0
		15×36	421200	362700	23400	0	0
`FLT-MI` NIST	192	12×17	137520	114600	9168	0	0
		9×22	85512	65322	6876	0	0
		7×29	53480	40110	5348	0	0
	256	12×22	183600	153000	12240	0	0
		9×29	110160	87210	9180	0	0
		8×33	89760	69360	8160	0	0
	384	18×22	579096	510156	27576	0	0
		14×29	364616	310996	21448	0	0
		12×33	275760	229800	18 384	0	0
	521	19×29	869440	770640	39520	0	0
		16×33	632320	549120	33280	0	0
		15×36	561600	483600	31200	0	0

TABLE 2.1 – Nombre d'opérations pour l'algorithme `FLT-MI` appliqué aux P standardisés du NIST et à des P aléatoires.

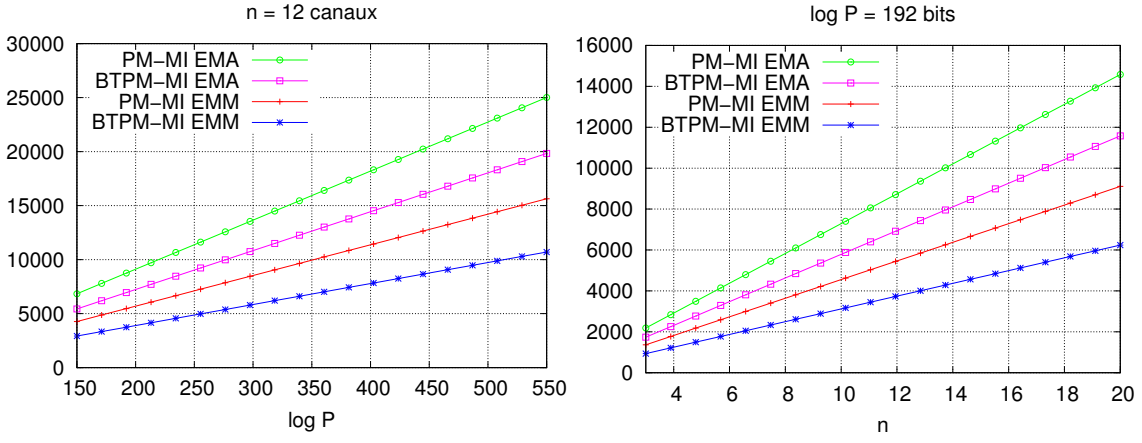

FIGURE 2.1 – Comparaison du nombre d'opérations `EMM` et `EMA` entre `PM-MI` et `BTPM-MI`, en fonction de $\log_2 P$ pour n fixé (à gauche) et en fonction de n pour $\log_2 P$ fixé (à droite).

Algo.	ℓ	$n \times w$	nombre d'opérations				
			EMM	EMA	cox-add	mod4-add	mod3
PM-MI	192	12×17	5474	8750	5474	5474	0
		9×22	4106	6562	4106	4106	0
		7×29	3193	5104	3193	3193	0
	256	12×22	7282	11656	7282	7282	0
		9×29	5461	8742	5461	5461	0
		8×33	4854	7771	4854	4854	0
	384	18×22	16487	26376	16487	16487	0
		14×29	12823	20514	12823	12823	0
		12×33	10991	17584	10991	10991	0
	521	19×29	23446	37522	23446	23446	0
		16×33	19744	31597	19744	19744	0
		15×36	18510	29622	18510	18510	0
BTPM-MI	192	12×17	3744	6994	2868	2868	269
		9×22	2808	5245	2151	2151	269
		7×29	2184	4079	1673	1673	269
	256	12×22	4988	9313	3821	3821	359
		9×29	3741	6985	2865	2865	359
		8×33	3325	6208	2547	2547	359
	384	18×22	11215	20928	8588	8588	538
		14×29	8723	16277	6680	6680	538
		12×33	7476	13952	5725	5725	538
	521	19×29	16055	29951	12293	12293	730
		16×33	13520	25222	10352	10352	730
		15×36	12675	23646	9705	9705	730

TABLE 2.2 – Nombre moyen d'opérations pour les algorithmes proposés PM-MI et BTPM-MI appliqués aux tailles de corps standards du NIST.

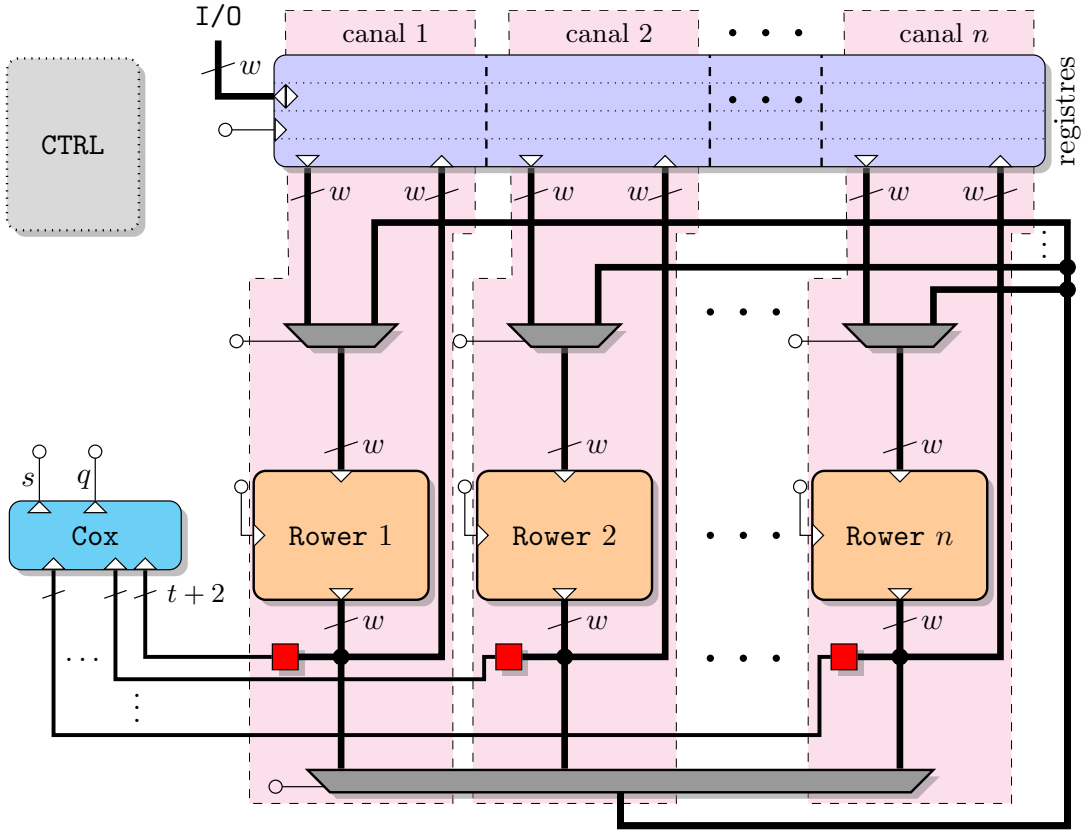

FIGURE 2.2 – Architecture globale `Cox-Rower` implantée similaire à celle proposée dans [52], avec `Cox` adapté à l'inversion `PM-MI`.

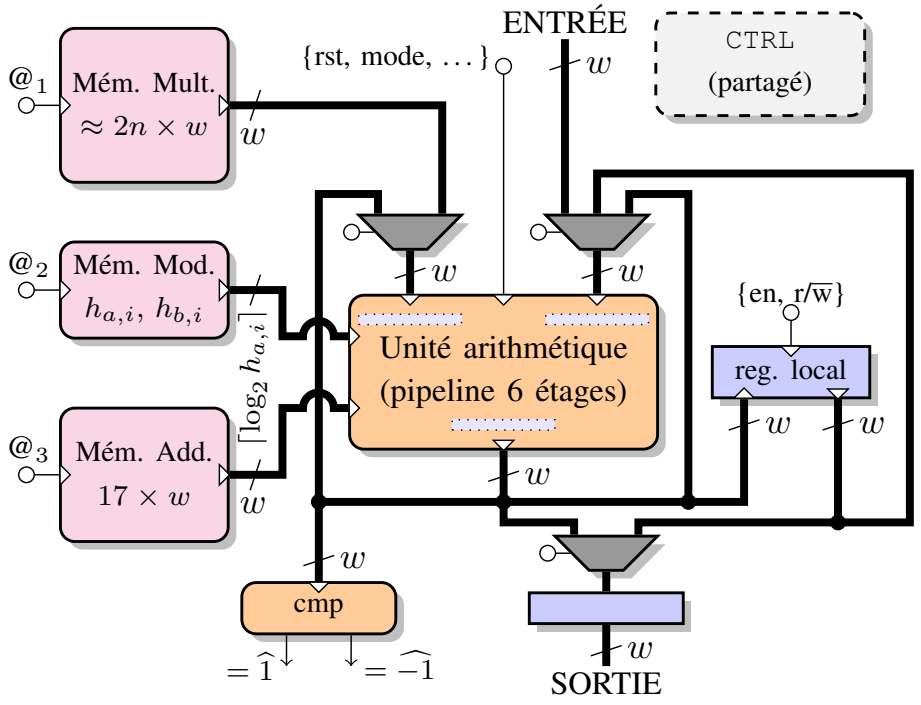

FIGURE 2.3 – Architecture du `Rower` implanté pour l'inversion `PM-MI`.

L'architecture globale est décrite dans la figure 2.2. En dehors du `Cox`, cette architecture globale est partagée par les 2 implantations, les autres composants étant ensuite spécialisés et optimisés en interne pour chacune des versions. Comme expliqué dans l'état de l'art 1.3.5, cette architecture `Cox-Rower` est complètement parallélisée, un `Rower` de w bits pour 2 canaux (un dans chaque base) pour effectuer des extensions de base. C'est aussi le cas pour notre algorithme car, même s'il n'utilise qu'une seule base, le reste des calculs pour ECC nécessitera 2 bases et des extensions de bases. Les signaux de contrôle, d'horloge et de reset ne sont représentés que partiellement dans la figure 2.2. Les lignes se terminant par les cercles blancs représentent des signaux de contrôle (—○).

La figure 2.3 présente la structure d'un élément `rower`. Chacun contient une unité arithmétique (AU) pour les calculs sur les opérandes de w bits, des registres locaux et des mémoires contenant les valeurs pré-calculées. L'unité arithmétique implantée est la même que celle décrite dans [52], dans la version pipelinée à 6 étages spécialement conçue pour les calculs ECC (voir section 1.3.5 et figure 1.9). La mémoire « Mém. Mod. » contient l'élément de chacune des bases correspondant à notre `Rower`. Ils sont stockés sous la forme $h_{a,i}$ et $h_{b,i}$ tels que $m_{a,i} = 2^w - h_{a,i}$ et $m_{b,i} = 2^w - h_{b,i}$. Notre implantation `PM-MI` utilise 21 valeurs pré-calculées stockées comme constantes dans les mémoires du `Rower`, et utilisées comme constantes multiplicatives (dans la mémoire « Mém. Mult. ») ou additives (dans « Mém. Add. »), en plus des valeurs stockées pour pouvoir effectuer une réduction modulaire de Montgomery RNS (nécessaires pour les calculs ECC).

La principale différence dans les `Rower`s entre la version `FLT-MI` et la version `PM-MI` sont les comparaisons sur la sortie de l'unité arithmétique (la figure 2.3 montre la version `PM-MI`). Celles-ci sont rajoutées pour tester la condition de la boucle principale.

L'unité `Cox` dans l'architecture originale est utilisée pour calculer la somme q définie

dans l'équation 1.13. Le `Cox` original est présenté dans la figure 1.8 de la section 1.3.5. C'est la même unité qui est implantée pour l'algorithme `FLT-MI`, c'est à dire juste un accumulateur qui somme n termes de t bits. Pour l'algorithme `PM-MI`, on utilise le `Cox` de façon intensive pour nos calculs modulaires. On l'a donc modifié pour que cette somme soit faite en un cycle. Dans nos implantations, le `Cox` n'était pas sur le chemin critique (situé au niveau de l'unité arithmétique), mais il se peut que pour un plus grand n ou un plus grand t il le soit. Cette somme pourrait alors être effectuée en 2 cycles par exemple ou 3. Le fait que le calcul soit fait en un cycle a permis d'avoir un contrôle un peu plus simple.

L'autre différence vient du fait que le nouveau `Cox` calcule aussi une somme modulo 4, et reçoit donc maintenant d'un côté les t bits de poids fort des différents restes et de l'autre leurs 2 bits de poids faible pour la somme $s = \left|\sum_{i=1}^{n} |\xi_{a,i}|_4\right|_4$. Dans la figure 2.2, les petits carrés représentent juste l'extraction et le routage des $t+2$ bits en provenance des w bits en sortie des `Rower`s. Les sorties du `Cox` sont s et q, et sont envoyées au contrôle qui calculera l'adresse @3, que ce soit pour corriger l'extension de base, ou effectuer le calcul adéquat dans la fonction $\widehat{\texttt{div2r}}$ de l'algorithme `PM-MI`.

Les valeurs globales de l'algorithme sont stockées dans les 4 registres globaux (en haut de la figure 2.2), contenant donc les valeurs RNS de n mots de w bits. Ces mots sont assignés à travers les n canaux, chaque canal ayant une entrée et une sortie spécifique. Les communications avec l'hôte sont effectuées à travers le port `I/O` de w bits (en haut à gauche) via ces registres.

Les deux versions de l'architecture ont été implantées sur un FPGA Virtex 5 à l'aide de l'outil Xilinx ISE 14.6. Pour $\ell = 192$ bits, le FPGA XC5VLX50T a été utilisé (il fait partie des petits FPGA de la famille Virtex 5), un FPGA plus grand (XC5VLX220) ayant été utilisé pour les autres tailles de corps. Les processus de synthèse et placement/routage ont été effectués pour des efforts standards d'optimisation en vitesse. Deux variantes ont été implantées, avec et sans blocs dédiés activés, afin de mesurer l'impact des blocs matériels DSP et BRAM (36Kb pour les FPGA Virtex 5). Nous avons implanté un certain nombre de bases avec différents compromis (n, w) pour les différentes tailles de corps. Pour $\ell = 192$, 256 et 521 bits, trois paires (n, w) ont été implantées. Pour avoir une courbe un peu plus précise sans avoir un temps de développement trop long, nous avons choisi d'implanter 4 paires en plus pour la taille 384, menant donc à un total de 7 pour ce corps.

Les résultats complets d'implantation sont présentés dans la table 2.3 pour les implantations avec blocs DSP et BRAM. La table 2.4 présente les résultats obtenus en désactivant les BRAM et les blocs DSP dans l'outil Xilinx ISE 14.6. On peut remarquer que dans certains cas, l'outil a quand même forcé l'utilisation de quelques blocs DSP malgré leur désactivation. Les résultats en terme de temps d'exécution dans la figure 2.4 et les résultats en terme de surface sont résumés dans les figures 2.5 et 2.6.

Algo.	ℓ	$n \times w$	Surface			Freq.	Nombre	Temps
			slices (FF/LUT)	DSP	BRAM	MHz	de cycles	μs
FLT-MI	192	12×17	2473 (2995/7393)	26	0	186	13416	72.1
		9×22	2426 (3001/7150)	29	0	185	11272	60.9
		7×29	2430 (3182/6829)	48	0	107	9676	90.4
	256	12×22	3303 (3960/9524)	38	0	190	15397	126.2
		9×29	3181 (3845/9444)	63	9	114	13093	114.9
		8×33	3065 (3838/8629)	56	8	105	12325	117.3
	384	22×18	4839 (6479/14642)	66	0	162	39971	246.7
		20×20	4972 (6037/15476)	62	0	180	37159	206.4
		18×22	4782 (5920/14043)	56	0	178	34359	193.0
		17×23	4818 (5830/14181)	53	0	180	32941	183.0
		14×29	5554 (5910/16493)	98	14	110	28416	258.3
		12×33	5236 (5710/15418)	84	12	107	25911	242.1
		10×40	11536 (7230/36503)	0	19	103	23099	224.3
	521	19×29	7344 (8113/21286)	128	19	120	51580	429.8
		16×33	6960 (7578/21282)	112	16	115	45334	394.2
		15×36	8006 (8550/27514)	128	15	90	43252	480.5
PM-MI	192	12×17	2332 (3371/6979)	26	0	187	1753	9.3
		9×22	2223 (3217/6706)	29	0	187	1753	9.3
		7×29	2265 (3336/6457)	48	0	120	1753	14.6
	256	12×22	2660 (3970/8638)	38	0	155	2333	15.0
		9×29	2934(4014/8117)	63	9	103	2333	22.6
		8×33	2688 (3838/7779)	56	16	105	2333	22.1
	384	22×18	5338 (6805/17620)	66	0	141	3518	25.0
		20×20	4201 (5977/12472)	62	0	148	3518	23.8
		18×22	4064 (5932/13600)	56	0	152	3518	23.1
		17×23	3956 (5848/12902)	53	0	150	3518	23.4
		14×29	4873 (6134/14347)	98	14	102	3518	34.4
		12×33	4400 (5694/12764)	84	24	103	3518	34.1
		10×40	9510 (7222/33451)	0	24	102	3518	34.5
	521	19×29	8044(8295/25983)	128	19	106	4750	44.8
		16×33	7275 (7553/22906)	112	32	108	4750	44.0
		15×36	8315 (8550/28438)	128	30	92	4750	51.6

TABLE 2.3 – Résultats d'implantation FPGA post placement/routage des algorithmes FLT-MI et PM-MI sur Xilinx Virtex 5 avec blocs DSP et BRAM activés dans ISE 14.6.

Algo.	ℓ	$n \times w$	Surface			Freq.	Nombre	Temps
			slices (FF/LUT)	DSP	BRAM	MHz	de cycles	μs
FLT-MI	192	12×17	4071 (4043/12864)	4	0	128	13416	104.8
		9×22	4155 (3816/13313)	4	0	122	11272	92.3
		7×29	4575 (3952/15264)	0	0	126	9676	76.7
	256	12×22	5334 (5092/17557)	4	0	122	15397	126.2
		9×29	5940 (5073/20097)	0	0	126	13093	103.9
		8×33	6416 (5116/20828)	0	0	104	12325	118.5
	384	22×18	8325 (7783/27330)	4	0	121	39971	330.3
		20×20	7826 (7847/27310)	4	0	120	37159	309.7
		18×22	7559 (7831/27457)	0	0	163	34359	210.7
		17×23	8104 (7547/27645)	4	0	123	32941	262.8
		14×29	9393 (7818/30536)	0	0	126	28416	225.5
		12×33	9888 (7640/31599)	0	0	107	25911	242.1
		10×40	10314 (7630/33911)	0	0	106	23099	217.9
	521	19×29	12072(10629/43347)	0	0	118	51580	437.1
		16×33	12306(10169/43732)	0	0	100	45334	453.3
		15×36	13438 (10324/45615)	0	0	87	43252	497.1
PM-MI	192	12×17	3899 (4212/12519)	4	0	150	1753	11.6
		9×22	3809 (3986/12782)	4	0	146	1753	12.0
		7×29	4341 (4107/14981)	0	0	141	1753	12.4
	256	12×22	4970(5037/16926)	4	0	123	2333	19.0
		9×29	5791 (5288/19660)	0	0	128	2333	18.2
		8×33	6100 (5292/20512)	0	0	107	2333	21.8
	384	22×18	9305 (8111/29280)	4	0	125	3518	28.1
		20×20	9046 (8158/29605)	4	0	127	3518	27.7
		18×22	7677 (8053/28306)	0	0	168	3518	20.9
		17×23	7536 (7467/25493)	4	0	121	3518	29.1
		14×29	9119(8113/30619)	0	0	127	3518	27.7
		12×33	9780 (7908/31902)	0	0	108	3518	32.5
		10×40	10111 (7495/33169)	0	0	100	3518	35.1
	521	19×29	14089 (11010/48858)	0	0	115	4750	41.3
		16×33	14397 (10489/48037)	0	0	105	4750	45.2
		15×36	15181 (10704/51922)	0	0	92	4750	51.6

TABLE 2.4 – Résultats d'implantation FPGA post placement/routage des algorithmes FLT-MI et PM-MI sur Xilinx Virtex 5 avec blocs DSP et BRAM désactivés dans ISE 14.6.

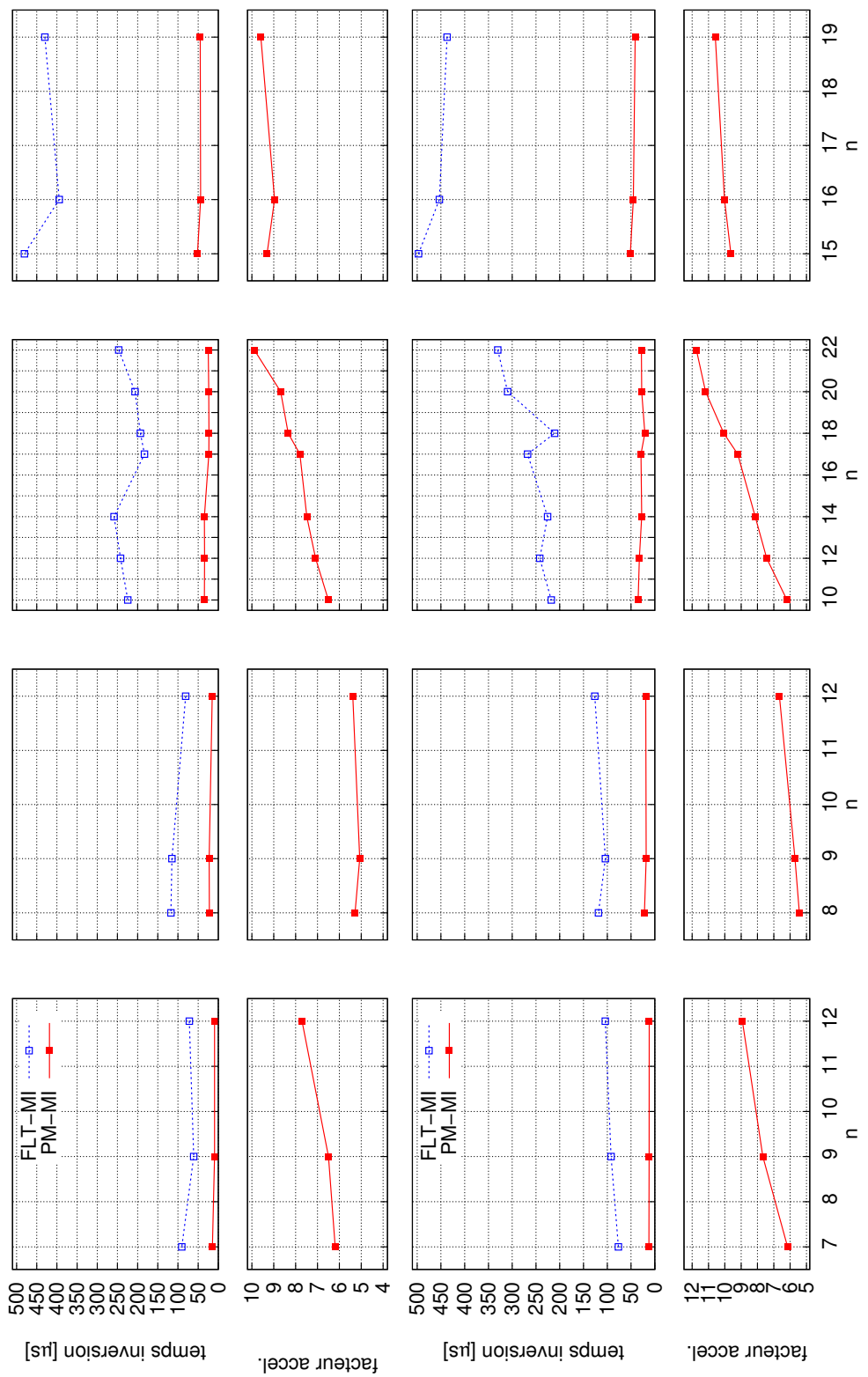

FIGURE 2.4 – Courbes des temps d'exécution et facteurs d'accélération des implantations FPGA des algorithmes `FLT-MI` et `PM-MI`, avec blocs DSP et BRAM activés (en haut) et désactivés (en bas) ; pour 4 tailles de corps du NIST et différentes paires (n, w).

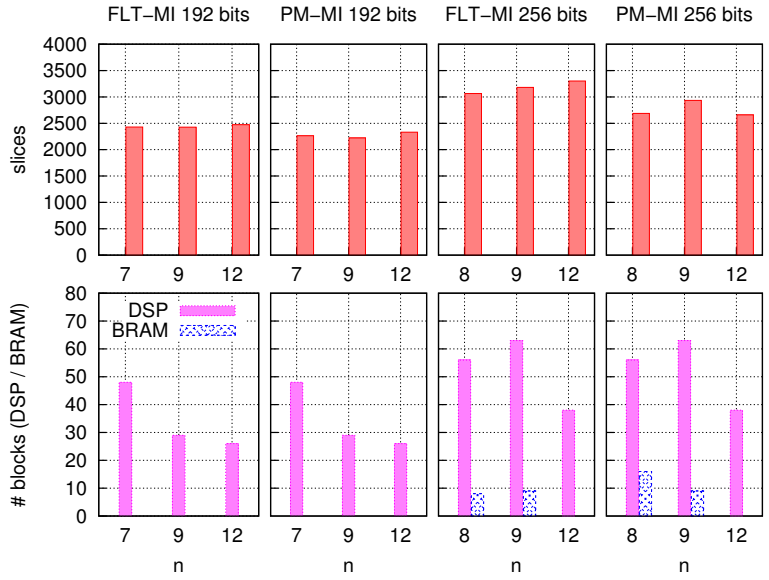

FIGURE 2.5 – Histogramme des surfaces occupées pour les implantations sur FPGA Virtex 5 pour les corps du NIST de 192 et 256 bits.

Avant d'analyser ces figures, nous présentons des arguments pour expliquer pourquoi nos implantations `FLT-MI` peuvent être utilisées comme référence afin de se comparer à l'état de l'art. Premièrement, nous avons implanté le `FLT-MI` avec un algorithme LSBF comme dans l'implantation ECC de l'état de l'art [52], tout en utilisant les astuces proposées (après la publication de [52]) pour l'exponentiation dans [48] pour accélérer les exponentiations RNS. L'algorithme a été implanté dans la version poids faibles en tête (algorithme 16) comme expliqué précédemment, afin de paralléliser le carré et la multiplication. Ensuite, il est difficile d'estimer l'efficacité d'architectures parallèles pour l'inversion modulaire en RNS dans l'état de l'art. Guillermin dans [52] estime qu'il y a jusqu'à 10 % de cycles d'attente dans les `Rower`s pour une multiplication scalaire complète en RNS (en comptant l'inversion). Ces cycles d'attente arrivent durant l'inversion et les conversions RNS/binaire et binaire/RNS. Nous avons considéré que ces cycles d'attente provenaient de l'inversion modulaire. En réalité, les opérations de conversions sont négligeables comparées à la complexité de l'inversion. De plus, l'algorithme d'exponentiation RNS utilisé dans [52] génère de nombreux cycles d'attente à cause des dépendances entre chaque itération de la boucle principale de l'algorithme d'exponentiation. On peut donc estimer que la plupart de ces cycles d'attente proviennent de l'inversion. En partant de cette hypothèse, le nombre de cycles d'attente avec le `FLT-MI` dans l'architecture de l'état de l'art [52] est d'environ 60 à 65 % du nombre total de cycles pour l'inversion. Nos implantations du `FLT-MI` ont elles entre 25 et 40 % de cycles d'attente (cela peut être facilement expliqué par le fait que dans nos implantations, le contrôle est dédié à notre opération d'inversion). On peut donc considérer que notre implantation du `FLT-MI` est un bon point de comparaison avec l'état de l'art.

La figure 2.4 permet de comparer les temps d'exécution obtenus et de mesurer l'écart entre les deux solutions en indiquant l'accélération obtenue entre le `FLT-MI` et le `PM-MI`. On peut donc voir que suivant les paramètres, on peut aller de 5 à 12 fois plus vite que la solution de l'état de l'art. On peut observer que le rapport des temps obtenus semble linéaire en n. C'est particulièrement clair sur le facteur d'accélération obtenu sur le corps

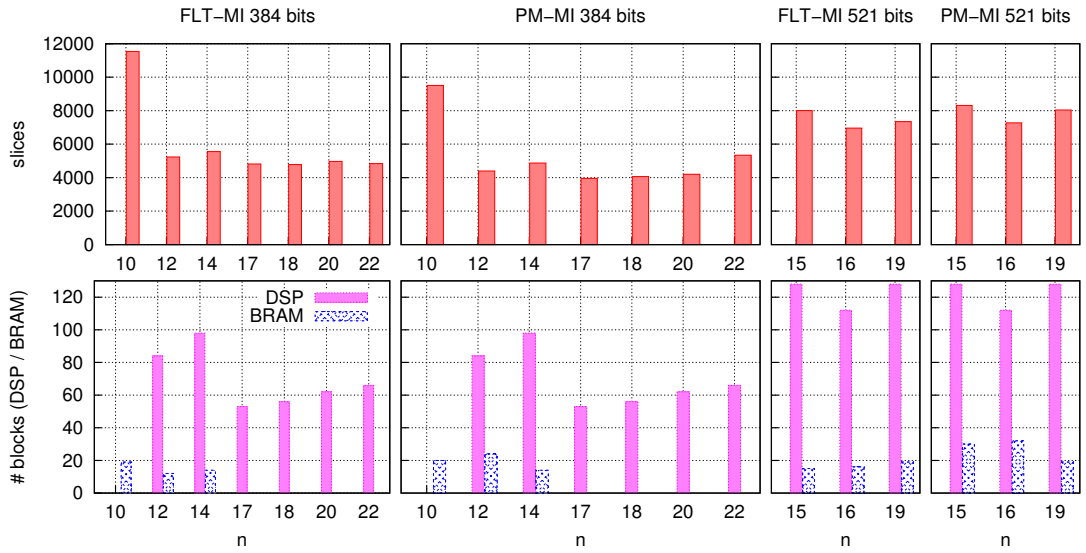

FIGURE 2.6 – Histogramme des surfaces occupées pour les implantations sur FPGA Virtex 5 pour les corps du NIST de 384 et 521 bits.

de 384 bits qui bénéficie de plus de points de mesure, en doublant n, on double quasiment le facteur d'accélération obtenu. C'est une conséquence directe des complexités des deux algorithmes détaillés en section 2.4. Pour rappel, la complexité du `PM-MI` est en $O(\ell \times n)$ contre $O(\ell \times n^2)$ pour le `FLT-MI`, le rapport des deux est bien en $O(n)$ en nombre d'opérations élémentaires.

On peut remarquer que pour le `PM-MI`, le nombre de cycles ne dépend pas des paramètres n ou w (on ne compte pas les cycles de lecture de l'opérande ou d'écriture de la sortie) mais seulement de la taille du corps. C'est tout simplement parce que l'on calcule sur une architecture complètement parallèle. Ainsi, il y a n unités arithmétiques pour $O(\ell \times n)$ opérations, on obtient bien alors un temps en $O(\ell)$ constant lorsque ℓ est fixé. Les variations du temps d'exécution sont donc dues aux variations de la fréquence d'horloge données par l'outil ISE.

A contrario, les implantations `FLT-MI` ont une complexité de $O(\ell \times n^2)$ opérations sur les canaux. Puisque nous avons implanté une architecture parallèle avec n `Rower`s pour autant de canaux, le nombre de cycles a une complexité de $O(\ell \times n)$, et dépend donc de n. Si nous choisissons un petit n, les éléments de la base seront grands (si $n = 4$ pour un corps de 384 bits par exemple, on se retrouve avec des moduli de presque une centaine de bits). Ce qui implique que les unités arithmétiques deviennent significativement plus lentes (la fréquence chute à cause d'aussi grandes unités). D'un autre côté, un grand n implique un plus grand nombre de cycles. Pour atteindre l'inversion `FLT-MI` la plus rapide possible, il faut réussir à trouver le meilleur compromis sur la taille des moduli. Cette taille dépend bien sûr du matériel sur lequel l'implantation est faite.

Ce compromis ne s'applique pas simplement au `FLT-MI`, mais à tout algorithme RNS utilisant des extensions de base, notamment les calculs sur les courbes elliptiques. Un autre point positif du `PM-MI` est qu'il ne dépend pas directement de ce compromis, on peut donc concevoir les unités pour avoir la multiplication scalaire la plus rapide qui soit, sans se

soucier de l'impact qu'il pourrait avoir sur le temps d'exécution de notre algorithme d'inversion. En d'autres termes, notre algorithme est très flexible au niveau du choix de la base RNS.

Pour conclure l'analyse de ces implantations, on peut observer que les deux algorithmes aboutissent à des coûts en surface proches pour presque tous les paramètres, comme on peut le voir sur les figures 2.5 et 2.6. On peut observer quelques petites différences, comme par exemple sur la figure 2.6 avec $n = 12$ et $\ell = 384$, l'implantation `PM-MI` a consommé moins de *slices* mais plus de BRAM que le `FLT-MI`. Les implantations étant extrêmement proches (en dehors du contrôle), il est normal que nous obtenions des résultats proches. Pour finir, on peut observer qu'il y a une chute de fréquence dans la table 2.3 entre les cas où $w < 25$ et les cas où $w \geq 25$. C'est dû au fait que les multiplieurs intégrés dans les blocs DSP soient des multiplieurs 25×18 bits pour entiers signés, ils permettent d'effectuer directement des multiplications de 24×17 bits sur les moduli. Le synthétiseur a réussi à mieux utiliser ces multiplieurs pour des tailles de mots $w < 25$, ce qui donne des fréquences plus élevées pour ces w là. Lorsque $w \geq 25$, des combinaisons plus complexes des multiplieurs sont effectuées, ce qui explique cette chute de fréquence.

2.6 Validation

Les algorithmes proposés ne sont pas ici prouvés en détail, mais de forts arguments sont présentés dans cette section pour leur validation.

Premièrement, la structure globale de l'algorithme 17 est la même que celle de l'algorithme d'Euclide binaire pour le calcul du pgcd avec plus-minus présenté dans [22]. Cela implique que si les opérations internes au RNS sont correctement effectuées, alors l'algorithme est correct. De plus, presque toutes les opérations nécessaires à l'algorithme sont juste des opérations classiques du RNS, additions et multiplications par des constantes. La fonction `mod4` détaillée précédemment est basée sur l'évaluation du CRT, utilisant la méthode de Kawamura *et al.* [64]. Nous nous assurons ensuite d'être dans le bon intervalle pour appliquer leur approximation, grâce à la représentation $\widehat{X}$.

Finalement, un total de 700 000 valeurs aléatoires (entre 0 et $3P$) ont été testées sur les modulo P-160, P-192, P-256, P-384 et P-521 (voir [91]) en utilisant Maple 15. Pour chacune des tailles implantées sur FPGA et chacun des deux algorithmes (`FLT-MI` et `PM-MI`), une paire (n, w) a été testée sur 10 000 valeurs aléatoires en simulation VHDL (les autres implantations ont été testées au travers de 1000 valeurs à inverser). Ces tests ont été effectués en générant des fichiers de test VHDL avec Sage 5.2, les simulations ont été effectuées avec ModelSim 6.6 et enfin la vérification des résultats effectuée par Sage elle aussi.

Quant à la version `BTPM-MI` (algorithme 18), il n'a pas encore été implanté en FPGA, sa validation a donc été effectuée seulement sur Maple 15 aussi avec 700 000 valeurs générées aléatoirement. Du côté de la validation théorique, on peut observer que les opérations internes de cet algorithme sont des adaptations de l'algorithme 17, et utilisent donc les mêmes arguments. Dans cet algorithme, comme dans l'algorithme d'Euclide étendu et sa variante binaire, on a toujours $V_1A + V_2P = V_3$ et $U_1A + U_2P = U_3$. C'est juste qu'en plus de gérer les divisions par 2 comme en binaire, on reporte aussi les divisions par 3. C'est aussi toujours cette condition qui fait qu'à la fin de l'algorithme, on a bien calculé l'inverse. Les opérations modulo 3, 6 et 12 permettent juste d'atteindre le résultat avec moins de

tours de boucles, puisqu'on divise plus rapidement que dans la version purement binaire.

2.7 Conclusion

Dans ce chapitre ont été présentés deux nouveaux algorithmes permettant de calculer l'inversion modulaire en RNS. Ces algorithmes, comme la plupart des algorithmes RNS utilisés en cryptographie, sont basés sur des idées qui ont été proposées dans le cas standard, mais où certaines difficultés techniques empêchaient leur réalisation en RNS. Le premier (`PM-MI`) est basé sur la version binaire de l'algorithme d'Euclide étendu, le second (`BTPM-MI`) propose une variante binaire-ternaire. Ces algorithmes ont la particularité de ne nécessiter qu'une seule base RNS, aucune extension de base n'étant requise.

Pour nos divers ensembles de paramètres cryptographiques, le nombre de multiplications élémentaires est réduit très significativement : on divise par un facteur 12 à 37 ce nombre pour le `PM-MI` et de 18 à 54 pour le `BTPM-MI`, comparé à l'algorithme basé sur le petit théorème de Fermat. Les algorithmes `PM-MI` et `FLT-MI` ont été implantés sur 32 ensembles de paramètres différents sur FPGA. Les résultats obtenus montrent des surfaces très similaires, et des implantations `PM-MI` qui sont 5 à 12 fois plus rapides que l'état de l'art.

Enfin, nous prévoyons d'implanter le nouvel algorithme binaire-ternaire sur FPGA, et d'étudier son coût en temps et surface comparé au `PM-MI` et d'intégrer ces algorithmes dans une implantation complète d'une multiplication scalaire ECC en RNS.

Chapitre 3

Décomposition et réutilisation d'opérandes pour la multiplication modulaire RNS

Ce chapitre est basé sur un travail en partie présenté lors de la conférence ASAP 2014 [20] sur la multiplication modulaire. Cet algorithme est, à notre connaissance, le premier à proposer une réduction du coût de la multiplication pour certains cas particuliers, comme le carré modulaire ou une multiplication par une constante. Pour des opérandes de n moduli, on obtient pour ces cas particuliers une complexité inférieure à $2n$ en terme de multiplications élémentaires (`EMM`) en utilisant des extensions sur des *bases réduites*, c'est-à-dire sur moins de moduli que l'état de l'art. Nous verrons que ces cas ou motifs de calcul particuliers se retrouvent dans les algorithmes d'exponentiation et dans les formules de courbes elliptiques.

3.1 Algorithme de multiplication modulaire RNS proposé

L'algorithme que nous avons proposé va décomposer le calcul de la multiplication modulaire en 2 parties. La première partie effectue des calculs indépendants pour chacun des deux opérandes, et la seconde combine les résultats de la première et, cette fois-ci, dépend des 2 opérandes. Lorsqu'on réutilise un des opérandes pour une nouvelle multiplication modulaire, la partie des calculs qui dépend uniquement de cette opérande va pouvoir être réutilisée, à la différence des algorithmes de l'état de l'art. En effet, dans l'état de l'art, un carré suivi d'une réduction modulaire en RNS coûte exactement le même prix qu'une multiplication modulaire. C'est la même chose si on multiplie par une constante ou plusieurs fois par la même valeur dans une séquence d'opérations.

Plus précisément, l'idée de notre algorithme est de décomposer dans un premier temps chacun des opérandes en 2 sous-valeurs. Ces sous-valeurs sont ensuite utilisées pour fabriquer une valeur sur $3n/2$ moduli au lieu de $2n$, c'est-à-dire un nombre de taille $\frac{3}{2}\log_2 P$ bits au lieu de $2\log_2 P$. Cette réduction du nombre total de moduli réduit le nombre de valeurs pré-calculées à stocker.

Pour les motifs comme le carré ou la multiplication par une constante, l'étape de décomposition est effectuée une seule fois, ce qui va permettre de gagner en terme de nombre d'opérations. Par exemple, lors d'une exponentiation avec l'algorithme « échelle de Mont-

gomery » (voir l'algorithme 22), un motif très utile est $A \leftarrow BC, D \leftarrow B^2$. Dans un tel cas, on remarque que la décomposition de B n'est effectuée qu'une seule fois, pour la première multiplication et pour le carré. On gagne donc ici 2 étapes de décomposition, par rapport à un motif $A \leftarrow BC, D \leftarrow EF$.

Nous allons maintenant détailler les différentes étapes de notre algorithme. Notre algorithme de multiplication (appelé `SPRR`, voir section 3.1.1) est divisé en 3 étapes : premièrement, la décomposition `Split`, ensuite la réduction partielle `PR` (pour *partial reduction*) pour obtenir des valeurs de $\frac{3}{2}\log_2 P$ bits, et enfin la réduction finale correspondant au dernier `R` de `SPRR`. Cette réduction utilise l'algorithme `MR` (réduction de Montgomery RNS) de l'état de l'art [48] (algorithme 15), pour obtenir une valeur de $\log_2 P + \varepsilon$ bits (typiquement $\varepsilon = 1$, 2 ou 3), c'est-à-dire une valeur inférieure à αP avec un petit α. L'algorithme complet est détaillé en section 3.1.2.

3.1.1 L'étape de décomposition

Afin d'expliquer ce qui est réalisé, nous allons d'abord détailler le flot de calcul qui est effectué dans les entiers (c'est-à-dire sans prendre en compte les détails de la représentation RNS). Toujours à des fins de compréhension, on va directement considérer ici que nous avons 3 bases, $\mathcal{B}_a$, $\mathcal{B}_b$ et $\mathcal{B}_c$, de $n/2$ moduli (3 « demi-bases »). On pourrait choisir d'autres paramètres pour la taille des bases, mais dans les cas que nous allons étudier dans ce chapitre, les meilleurs résultats sont obtenus dans ce cas précis.

Tout d'abord, durant l'étape de décomposition, les opérandes X et Y vont être décomposés en un couple reste/quotient par M_a. L'algorithme 20 présente comment est faite cette décomposition `Split`. Après l'algorithme `Split`, on obtient en sortie $X = K_x M_a + R_x$ et $Y = K_y M_a + R_y$. La première extension de base `BE` (ligne 1) convertit le reste de l'entrée X modulo M_a (noté R_x) dans les bases $\mathcal{B}_b$ et $\mathcal{B}_c$. Il y a ici une réduction implicite par M_a dans la base $\mathcal{B}_a$. À partir du reste, il est facile de calculer le quotient K_x ligne 2 dans les bases $\mathcal{B}_b$ et $\mathcal{B}_c$. Les lignes 3 à 5 ne sont utiles que si on utilise l'extension de base de Kawamura [64], et seront expliquées dans le paragraphe suivant. Si on utilise une extension de base via MRS [12, 120], ces lignes sont inutiles. On remarque que X est une valeur de ℓ bits puisque c'est un élément modulo P, donc K_x est une valeur de $\ell/2$ bits. K_x peut ainsi être extrait à partir d'une seule des deux demi-bases $\mathcal{B}_b$ ou $\mathcal{B}_c$ (ligne 6). Dans l'algorithme présenté, on suppose qu'on fait l'extension de base à partir de $\mathcal{B}_b$. Il se peut en fait que X soit de taille $\ell + \varepsilon$ bits en étant dans le domaine de Montgomery, mais ça ne va pas changer les résultats obtenus. À la fin de l'algorithme, nous avons donc K_x et R_x dans les 3 demi-bases.

Si nous utilisons l'algorithme d'extension de base de Kawamura *et al.* [64], qui est plus efficace que l'extension de base via MRS [120], alors l'approximation qu'elle implique lors de la première extension de base ligne 1 de l'algorithme 20 peut retourner R_x, ou la valeur approchée $R'_x = R_x + M_a$. Ceci requiert d'agrandir un peu la valeur de M_b. Dans le cas d'une d'approximation de R_x par $R'_x = R_x + M_a$, on remarque que le calcul ligne 2 de l'algorithme 20 donne $K'_x = K_x - 1$. Si $K_x \geq 1$, alors $K'_x \geq 0$, et il n'y a aucun impact sur le reste de l'algorithme, car malgré l'approximation on a quand même $X = K'_x M_a + R'_x$. Par contre, si $K_x = 0$ on obtient $K'_x = -1$, qui ne satisfait plus les hypothèses du théorème de Kawamura *et al.* (théorème 5). C'est le seul cas spécial à traiter ici. Pour pallier ce problème, il suffit de tester si $K'_x = -1$ dans $\mathcal{B}_b$ et si c'est le cas, de corriger avec $K'_x = 0$

et $R'_x = R_x - M_a$ (lignes 3-5 de l'algorithme 20). Si l'architecture implantée utilise l'inversion modulaire `PM-MI` présentée au chapitre 2, ce test d'égalité avec -1 est déjà présent en matériel en sortie des `Rower`s. Dans la suite de ce chapitre, sauf mention contraire, nous ne distinguerons plus K_x de K'_x (respectivement R_x de R'_x).

La seconde extension de base ligne 6, elle, doit être exacte, sous peine d'obtenir un résultat faux. Pour rappel, il faut pour cela que $(1-\sigma_0)M_b > K_x$, ce qui peut se traduire par $(1-\sigma_0)M_b > \frac{\alpha P}{M_a}$. Cela implique donc une contrainte sur la taille de M_b et sur la base $\mathcal{B}_b$ pour satisfaire le théorème 4 (cas exact de Kawamura *et al.*). Les propositions 1 et 2 de la section 3.1.2 intègrent ces contraintes, qui sont ensuite expliquées dans la section 3.1.3 de démonstration.

Algorithme 20: Étape de décomposition (`Split`).

Entrée : $\overrightarrow{X_{a|b|c}}$, $X < \alpha P$
Pré-calcul : $\overrightarrow{\left(M_a^{-1}\right)_{b|c}}$
Sorties : $\overrightarrow{(K_x)_{a|b|c}}$, $\overrightarrow{(R_x)_{a|b|c}}$ avec
$\overrightarrow{X_{a|b|c}} = \overrightarrow{(K_x)_{a|b|c}} \times \overrightarrow{(M_a)_{a|b|c}} + \overrightarrow{(R_x)_{a|b|c}}$

1 $\overrightarrow{(R_x)_{b|c}} \leftarrow \texttt{BE}\left(\overrightarrow{(R_x)_a}, \mathcal{B}_a, \mathcal{B}_{b|c}\right)$
2 $\overrightarrow{(K_x)_{b|c}} \leftarrow \left(\overrightarrow{X_{b|c}} - \overrightarrow{(R_x)_{b|c}}\right) \times \overrightarrow{\left(M_a^{-1}\right)_{b|c}}$
3 **si** $\overrightarrow{(K_x)_{b|c}} = \overrightarrow{-1}$ **alors**
4 $\quad \overrightarrow{(K_x)_{b|c}} \leftarrow \overrightarrow{0}$ /*correction de l'erreur d'extension de base */
5 $\quad \overrightarrow{(R_x)_{b|c}} \leftarrow \overrightarrow{(R_x)_{b|c}} - \overrightarrow{(M_a)_{b|c}}$
6 $\overrightarrow{(K_x)_a} \leftarrow \texttt{BE}\left(\overrightarrow{(K_x)_b}, \mathcal{B}_b, \mathcal{B}_a\right)$
7 **retourner** $\overrightarrow{(K_x)_{a|b|c}}$, $\overrightarrow{(R_x)_{a|b|c}}$

Pour réduire le nombre de calculs, on peut observer que la multiplication ligne 2 de l'algorithme 20 par $\overrightarrow{\left(M_a^{-1}\right)_{b|c}}$ peut être combinée avec celle par $\overrightarrow{\left(T_b^{-1}\right)_b}$, dans la première ligne de l'algorithme 14 d'extension de base (pour la seconde extension de base de l'algorithme 20). Nous sauvons ainsi n_b `EMM`. Comme indiqué dans la table 3.2, on pré-calcule alors $\overrightarrow{\left(M_a^{-1}T_b^{-1}\right)_b}$. Cette table inclut les pré-calculs des `BE` utilisées dans `Split`.

Finalement, le coût de l'algorithme `Split` est $(n_a + n_a(n_b+n_c)) + (n_b + n_b n_a) + n_c$ `EMM`, pour les deux extensions de base. Dans notre cas, c'est-à-dire avec $n_a = n_b = n_c = n/2$, le coût total est alors $\frac{3}{4}n^2 + \frac{3}{2}n$ `EMM`. Pour conclure, on peut voir l'étape `Split` comme une sorte de décomposition à la manière de Karatsuba-Ofman [63]. Dans le cas de Karatsuba-Ofman, on effectue ensuite 3 multiplications au lieu de 4. Dans notre cas, nous allons voir que nous allons réduire une valeur représentée sur $\frac{3}{2}n$ au lieu de $\frac{4}{2}n = 2n$ moduli. L'analogie s'arrête ici, la multiplication de Karatsuba-Ofman ne semblant pas s'appliquer (de façon efficace) à la représentation RNS, car c'est une représentation des nombres non positionnelle.

3.1.2 Algorithme de multiplication modulaire SPRR

Notre méthode complète est présentée dans l'algorithme 21 et illustrée sur la figure 3.1. On suppose que les entrées de la multiplication modulaire sont inférieures à αP.

Algorithme 21: Multiplication modulaire proposée SPRR.

Entrées : $\overrightarrow{X_{a|b|c}}$, $\overrightarrow{Y_{a|b|c}}$, avec $X, Y < \alpha P$
Pré-calcul : $D = |M_a^{-1}|_P$
Sortie : $\overrightarrow{V_{a|b|c}}$ avec $V \equiv \left|XYM_a^{-1}M_b^{-1}\right|_P$ et $V < \alpha P$
1 $(\overrightarrow{(K_x)_{a|b|c}}, \overrightarrow{(R_x)_{a|b|c}}) \leftarrow \texttt{Split}(\overrightarrow{X_{a,b,c}})$
2 $(\overrightarrow{(K_y)_{a|b|c}}, \overrightarrow{(R_y)_{a|b|c}}) \leftarrow \texttt{Split}(\overrightarrow{Y_{a,b,c}})$
3 $\overrightarrow{U_{a|b|c}} \leftarrow \texttt{PR}\left(\overrightarrow{(K_x)_{a|b|c}}, \overrightarrow{(R_x)_{a|b|c}}, \overrightarrow{(K_y)_{a|b|c}}, \overrightarrow{(R_y)_{a|b|c}}, D\right)$
4 $\overrightarrow{V_{a|b|c}} \leftarrow \texttt{MR}(\overrightarrow{U_b}, \overrightarrow{U_{a|c}})$
5 **retourner** $\overrightarrow{V_{a|b|c}}$

Après les deux étapes Split aux lignes 1 et 2, une opération PR de réduction partielle est utilisée (et détaillée juste ensuite) qui permet d'obtenir une valeur sur $\frac{3}{2}n$ moduli, c'est-à-dire de taille $\frac{3}{2}\log_2 P$. Enfin, ligne 4, on effectue une dernière réduction en utilisant l'algorithme 15 de réduction de Montgomery en RNS sur ces bases réduites. Après les décompositions lignes 1 et 2, on a :

$$|X\,Y|_P = \left|K_xK_yM_a^2 + (K_xR_y + K_yR_x)M_a + R_xR_y\right|_P. \tag{3.1}$$

Afin de faire une réduction partielle à partir de l'équation 3.1, nous allons supposer la relation qui suit entre la base $\mathcal{B}_a$ et P.

Hypothèse 1 (H_1)**.** *Soit $D = |M_a^{-1}|_P$, c'est-à-dire l'inverse de M_a modulo P. On suppose alors qu'il existe μ, avec μ très petit (typiquement 1, 2 ou 3) tel que $\mu P + 1 = M_a \times D$.*

Sous cette hypothèse H_1, nous définissons U comme

$$U = K_xK_yM_a + (K_xR_y + K_yR_x) + R_xR_yD \equiv |X\,Y\,D|_P. \tag{3.2}$$

En choisissant $n_a = n/2$, les tailles de M_a, R_x, K_x et D sont très proches de $\ell/2$ bits et on a $\log_2 U \approx \frac{3}{2}\log_2 P$ sous H_1. Ainsi, puisque $n_a = n_b = n_c = n/2$, il suffit de calculer U sur les bases $\mathcal{B}_a$, $\mathcal{B}_b$ et $\mathcal{B}_c$ ($\frac{3n}{2}$ moduli en tout). Plus précisément, en posant $C(\alpha,\mu) = \frac{\alpha^2}{\mu}D + 4\alpha + 4\mu M_a$, on a :

$$0 \leqslant U < C(\alpha,\mu)P + 4M_a. \tag{3.3}$$

Ce résultat sera prouvé en section 3.1.3. Pour bien comprendre ce qu'implique cette inégalité, il faut tout d'abord voir qu'on peut négliger 4α dans l'expression de $C(\alpha,\mu)$, car en pratique α est pris petit (typiquement $\alpha = 2$ ou 3, cf. [10, 52, 64]). Ensuite, l'impact du terme $4M_a$ est très limité, car pour notre algorithme $\log_2 M_a < \frac{\ell}{2}$. Pour résumer, U est borné par un entier Γ de la forme $\Gamma = (\gamma_1 D + \gamma_2 M_a)P$, avec γ_1 et γ_2 des petits entiers.

La première conclusion est que si $\log_2 M_a \approx \frac{\ell}{2}$, on a $\log_2 D \approx \frac{\ell}{2}$ et donc $\log_2 \Gamma \approx \frac{3\ell}{2}$ (on rappelle que $\lceil \log_2 P \rceil = \ell$). La deuxième conclusion est que la taille de U est directement

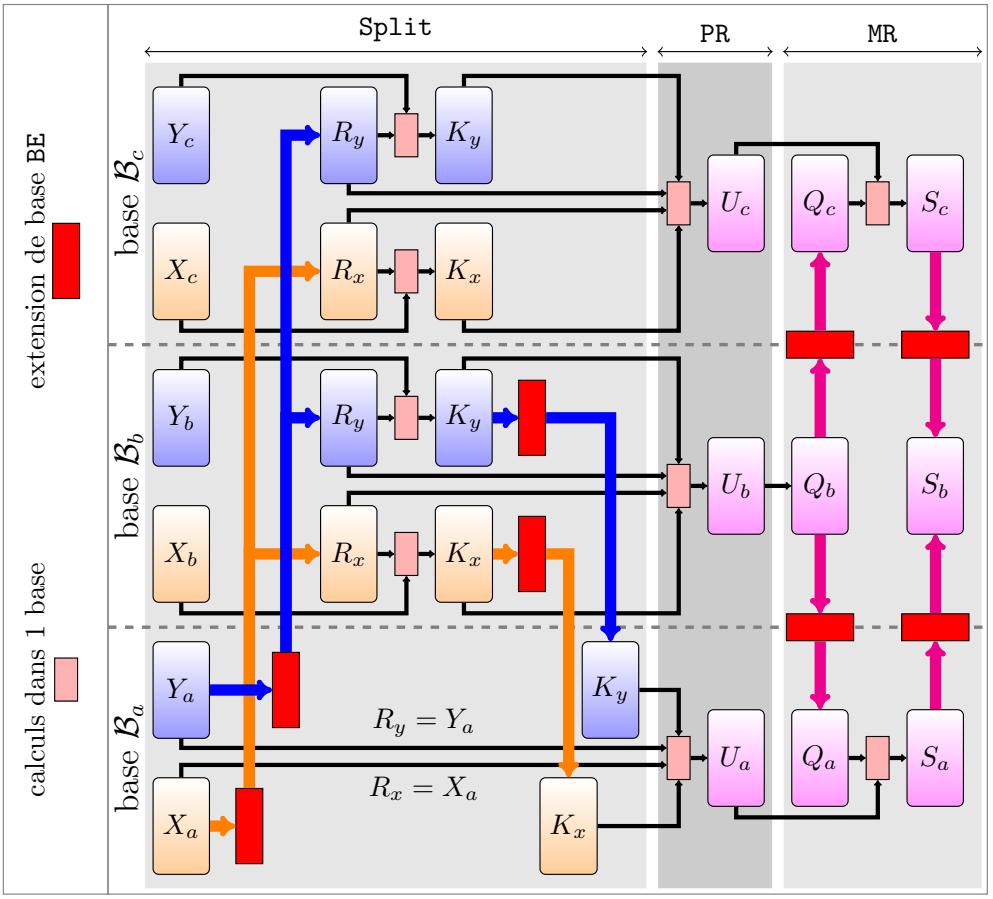

FIGURE 3.1 – Flot de calcul dans l'algorithme SPRR.

liée aux tailles de M_a et D, et plus précisément à $\max(\lceil \log_2 M_a \rceil, \lceil \log_2 D \rceil)$.

Nous allons maintenant justifier le choix d'une base $\mathcal{B}_a$ avec $n/2$ moduli et la nécessité de la condition H_1. Dans notre algorithme, nous voulons minimiser la taille de U, autrement dit avoir une réduction partielle la plus efficace possible. Ainsi, la réduction finale MR est effectuée sur le moins de moduli possibles, nous permettant de réduire le coût des extensions de bases. Il nous faut donc minimiser $\max(\lceil \log_2 M_a \rceil, \lceil \log_2 D \rceil)$. À cause du fait que $D = |M_a^{-1}|_P$, il est impossible que M_a et D aient une taille plus petite que $\ell/2$ tous les deux à la fois, tout simplement parce que $M_a D > P$ et donc $\log_2 M_a + \log_2 D > \ell$. Ainsi, la borne sur U est proche d'être minimale lorsque $\log_2 M_a \approx \log_2 D \approx \ell/2$, ce qui explique notre choix de départ sur le nombre de moduli de $\mathcal{B}_a$ et la nécessité de la condition H_1.

Remarque. Si on ne s'autorisait aucune condition liant la base $\mathcal{B}_a$ et P, alors D serait un élément quelconque de $\mathbb{F}_P$, de taille ℓ. Dans ce cas, on aurait $\log_2 U > 2\ell$ et la dernière étape de l'algorithme MR coûterait exactement une réduction modulaire de Montgomery en RNS de l'état de l'art. Avec en plus le coût des autres étapes, notre algorithme serait finalement bien plus coûteux que l'état de l'art.

L'étape PR de l'algorithme 21 calcule U à partir de l'équation 3.2. Cette opération coûte au premier abord 6 multiplications RNS sur chacun des $\frac{3n}{2}$ moduli (c.-à-d. $9n$ EMM). En utilisant la méthode de Karatsuba-Ofman sur l'équation 3.2, on obtient :

$$\begin{aligned} U =& K_xK_yM_a + (K_xK_y + R_xR_y - (K_x - R_x)(K_y - R_y)) + R_xR_yD \\ =& K_xK_y(M_a + 1) + R_xR_y(D + 1) - (K_x - R_x)(K_y - R_y) \quad . \end{aligned} \tag{3.4}$$

On a donc plus que 5 multiplications par moduli menant à $7.5n$ `EMM`. On remarque que dans la première base $\mathcal{B}_a$, la multiplication par $(M_a + 1)$ n'en est pas vraiment une car cela revient à multiplier par 1. Cela réduit le coût d'une `EMM` par élément de $\mathcal{B}_a$, on obtient $7n$ `EMM` en tout.

Enfin, le calcul de U peut être réduit dans certains cas particuliers. Par exemple, si nous calculons un carré, alors l'équation 3.2 ne requiert plus que 3 multiplications dans $\mathcal{B}_a$ (K_xR_x et $(R_xR_y)D$), ce qui mène à un coût total de $6.5n$ pour `PR`.

Dans la séquence d'opérations $(|XY|_P, |XZ|_P)$, les calculs $K_x(M_a + 1)$ et $R_x(D + 1)$ sont communs aux deux multiplications, ces calculs peuvent donc être factorisés. Dans ce cas, il ne reste plus que 3 `EMM` par canal sur $\mathcal{B}_a$, $\mathcal{B}_b$ et $\mathcal{B}_c$ d'après l'équation 3.4, menant à un coût total de $4.5n$ pour `PR`.

Dans le cas d'une multiplication par une constante C, le coût peut être réduit à $3n$ en calculant directement

$$U = K_x|C|_P + R_x|CD|_P \quad .$$

Pour ce dernier cas, il n'y a plus que 2 multiplications par canal, on obtient bien $3n$ `EMM` au total.

La dernière étape de l'algorithme applique la réduction modulaire RNS de l'état de l'art (algorithme 15), mais seulement sur $\frac{3}{2}n$ moduli. La base de départ est la base $\mathcal{B}_b$ et celle d'arrivée est $\mathcal{B}_{a|b}$, la concaténation de $\mathcal{B}_a$ et $\mathcal{B}_c$. Le coût de cette dernière étape est $2n_b(n_a + n_c) + 2(n_a + n_c) + nb = n^2 + \frac{5}{2}n$ `EMM`. Ce coût prend en compte les améliorations proposées par Guillermin [52] et Gandino *et al.* [48] qui sont basées sur la factorisation de certains pré-calculs. L'amélioration de Gandino *et al.* nécessitant un changement de la représentation RNS dans une des bases ne semble pas utilisable ici de façon efficace.

La proposition 1 résume toutes les conditions requises pour appliquer notre multiplication. Sa preuve est donnée dans la section 3.1.3.

Proposition 1 (sous l'hypothèse H_1)**.** *Soit P un grand entier, premier avec les 3 bases RNS notées $\mathcal{B}_a$, $\mathcal{B}_b$ et $\mathcal{B}_c$. Si $M_b > \frac{1}{\alpha-2}(C(\alpha, \mu) + 1)$, $M_c > \frac{(\alpha-2)P}{M_a}$ et $X, Y < \alpha P$ alors, en sortie de l'algorithme 21, on a $\overrightarrow{V_{a|b|c}} = |XYM_a^{-1}M_b^{-1}|_P$ avec $V < \alpha P$.*

Ce théorème peut être réécrit facilement pour une autre hypothèse H_2, très proche de H_1.

Hypothèse 2 (H_2)**.** *Soit $D = |-M_a^{-1}|_P$, c'est-à-dire l'inverse de $(-M_a)$ modulo P. On suppose alors qu'il existe μ, avec μ très petit (typiquement 1, 2 ou 3) tel que $\mu P - 1 = M_a \times D$.*

L'hypothèse H_2 sera utilisée pour des applications du logarithme discret dans les corps finis comme les cryptosystèmes de type Diffie-Hellman. Sous l'hypothèse H_2, nous redéfinissons U avec

$$U = K_xK_yM_a + (K_xR_y + K_yR_x) - R_xR_yD + M_aP \equiv |XY(-D)|_P \, .$$

On a rajouté le terme M_aP à l'expression de U pour s'assurer qu'elle soit bien positive. Cette variation H_2 de l'hypothèse H_1 définit aussi une nouvelle version de la proposition. La seule différence provient de la condition sur la taille de M_b.

Proposition 2 (sous l'hypothèse H_2)**.** *Soit P un grand entier, premier avec les 3 bases RNS notées $\mathcal{B}_a$, $\mathcal{B}_b$ et $\mathcal{B}_c$. Si $M_b > \frac{1}{\alpha-2}\left(C(\alpha,\mu)+M_a+1\right)$, $M_c > \frac{(\alpha-2)P}{M_a}$ et $X,Y < \alpha P$ alors, en sortie de l'algorithme 21, on a $\overrightarrow{V_{a|b|c}} = |XYM_a^{-1}M_b^{-1}|_P$ avec $V < \alpha P$.*

Nous ne traitons pas dans cette section du coût de l'algorithme complet car nous verrons dans la section 3.2 que ce coût dépend fortement de la séquence de calculs dans laquelle on l'utilise. Ceci est bien illustré dans la figure 3.1, où les étapes `Split` des 2 entrées sont complètement indépendantes. Ces étapes `Split` peuvent être factorisées avec d'autres appels de `SPRR` réutilisant l'un des opérandes. Dans le cas d'un carré, un seul `Split` est effectué. L'étude du coût de l'algorithme n'a d'intérêt que dans les motifs d'opérations utilisés dans les différentes applications.

Pour conclure la présentation de notre algorithme, on peut remarquer que pour pouvoir enchaîner les multiplications, il nous faut convertir les entrées dans le domaine de Montgomery. Les entrées sont ainsi de la forme $X' = |XM_aM_b|_P$ et $Y' = |YM_aM_b|_P$, et l'algorithme `SPRR` retourne alors $|XYM_aM_b|_P$. On garde ainsi la même représentation que l'état de l'art, mais sur moins de moduli.

3.1.3 Preuve des propositions 1 et 2

Nous allons démontrer dans cette sous-section les bornes des propositions 1 et 2. Les démonstrations étant très similaires pour les deux propositions, on va tout d'abord se focaliser sur le cas de la proposition 1, c'est-à-dire en considérant H_1. Pour commencer, nous allons étudier la taille de U, sortie du calcul de l'étape `PR`. Cette étape fait le lien entre les extensions de base de l'étape `Split` et celles de la réduction finale `MR` (ligne 4 de l'algorithme 21).

Par définition du reste de la division euclidienne, on a $0 \leqslant R_x, R_y < M_a$. De plus, sous l'hypothèse H_1 on a $M_aD = \mu P + 1$, et puisque $X < \alpha P$ alors :

$$0 \leqslant K_x \leqslant \frac{\alpha P}{M_a} \leqslant \frac{\alpha P}{\mu P + 1} D < \frac{\alpha}{\mu} D \quad .$$

Comme expliqué dans la section 3.1.1, si l'approximation de l'extension de base de Kawamura *et al.* [64] nous fait commettre une approximation dans le calcul, on obtiendra à la sortie de `Split` les valeurs $R'_x = R_x + M_a$ et $K'_x = K_x - 1$. Nous n'avons ici pas de moyen de détecter si K_x ou K'_x a été calculé (sauf pour le cas précis où $K'_x = -1$), on va donc considérer des encadrements plus larges $-1 \leqslant K_x, K_y < \frac{\alpha}{\mu} D$ et $0 \leqslant R_x, R_y < 2M_a$. On obtient alors la borne supérieure suivante sur U :

$$\begin{aligned} U &= K_x(K_yM_a) + K_xR_y + K_yR_x + R_x(R_yD) \\ &< \frac{\alpha}{\mu} D(\alpha P) + 2\alpha P + 2\alpha P + 4M_a\left(\mu P + 1\right) \\ &< \left(\frac{\alpha^2}{\mu} D + 4\alpha + 4\mu M_a\right) P + 4M_a \\ &\leqslant C(\alpha,\mu)P + 4M_a \quad , \end{aligned}$$

où $C(\alpha,\mu) = \left(\frac{\alpha^2}{\mu} D + 4\alpha + 4\mu M_a\right)$ permet de simplifier l'écriture. On rappelle que l'on se place sous l'hypothèse H_1, et que $\log_2 U \approx \frac{3\ell}{2}$.

Pour que $U \equiv |XYD|_P$, il ne faut pas qu'une réduction implicite se produise. Tout d'abord, nous pouvons remarquer que U n'est jamais négatif car, en sortie de `Split`, K_x et R_x ne sont jamais négatifs. Nous rappelons que les effets générés par l'approximation due à l'utilisation de l'extension de base de Kawamura *et al.* sont gérés à l'intérieur de la fonction `Split`. Pour qu'il n'y ait pas de réduction implicite, il faut aussi que le produit de tous les moduli soit supérieur à U. Ceci explique que l'on ait besoin de 3 demi-bases, et pas seulement 2, afin de disposer des $\frac{3n}{2}$ moduli nécessaires à la représentation de U. En effet, la troisième base $\mathcal{B}_c$ n'est en réalité pas nécessaire pour réaliser les étapes `Split` et `PR`. Par contre, elle est requise pour effectuer notre dernière étape, pour avoir le nombre minimal de moduli requis pour représenter U. C'est pourquoi il faut que les calculs soient aussi réalisés sur cette base. On choisit donc M_b et M_c tels que $M_aM_bM_c > C(\alpha,\mu)P+4M_a$.

On va tout d'abord fixer les conditions sur M_b, puis en déduire la condition sur M_c. Nous commençons par M_b car c'est M_b qui va jouer le rôle de la première base dans l'algorithme 15 `MR` et qui va impacter la taille de la sortie de l'algorithme `SPRR`. Nous souhaitons que les éléments retournés par l'algorithme `SPRR` soient inférieurs à αP pour être réutilisés ensuite comme une entrée dans un autre appel de `SPRR`. L'algorithme 15 `MR` est ici appliqué avec $\mathcal{B}_b$ comme première base et $\mathcal{B}_{a|c}$ comme seconde base. On rappelle que la base $\mathcal{B}_{a|c}$ est la concaténation des bases $\mathcal{B}_a$ et $\mathcal{B}_c$, et a deux fois plus d'éléments que $\mathcal{B}_b$. Le résultat de `SPRR` est le résultat de l'algorithme `MR` appliqué ligne 4, c'est-à-dire $\frac{U+Q_{a|c}P}{M_b}$(voir algorithme 15). En partant de $\frac{U+Q_{a|c}P}{M_b} < \alpha P$ on obtient :

$$0 \leqslant \frac{U}{M_b} < \left(\alpha - \frac{Q_{a|c}}{M_b}\right) P < (\alpha - 2)\, P \quad , \tag{3.5}$$

car $Q_{a|c} < 2M_b$. Cela vient du fait que $Q_{a|c}$ est la sortie de la première extension de base (ligne 2 de l'algorithme 15). On rappelle que dans le cas de la première extension de base de l'algorithme 15, la résultat n'est pas forcément exact, c'est pourquoi on considère $Q_{a|c} < 2M_b$ et non pas seulement $Q_{a|c} < M_b$. On déduit finalement une condition suffisante sur M_b pour satisfaire l'équation 3.5, en utilisant la borne sur U. En remarquant que $\frac{4M_a}{P} < 1$, on peut choisir M_b tel que :

$$M_b > \frac{1}{\alpha - 2}\,(1 + C(\alpha,\mu)) > \frac{1}{\alpha - 2}\left(\frac{4M_a + C(\alpha,\mu)P}{P}\right) > \frac{U}{(\alpha-2)P} \quad . \tag{3.6}$$

On remarque qu'en appliquant cette condition, M_b est forcément plus grand que $4M_a$, ce qui permet de remplir directement la condition pour effectuer l'étape `Split` avec l'extension de base de Kawamura *et al.* [64].

Pour conclure, nous allons donner une version simplifiée de la condition sur M_c, en supposant l'équation 3.6. Il faut que le produit de tous les éléments des 3 demi-bases, c'est-à-dire $M_aM_bM_c$ soit supérieur à U. On peut donc transcrire cette condition par :

$$M_aM_bM_c > C(\alpha,\mu)P + 4M_a\,.$$

Pour simplifier le choix de $\mathcal{B}_c$ et donc de M_c, on peut directement utiliser la condition sur M_b (équation 3.6) en lieu et place de M_b, nous menant à la condition $Mc > (\alpha-2)\frac{P}{M_a}$. On vérifie alors bien :

$$M_aM_bM_c \quad > (\alpha-2)PM_b > C(\alpha,\mu)P + 4M_a\,.$$

Nous avons ici démontré que les bornes sur M_b et M_c étaient suffisantes pour obtenir le résultat souhaité à la fin de l'algorithme SPRR, sous l'hypothèse H_1. Si maintenant nous considérons H_2, très peu de changements sont à effectuer. En effet, H_2 implique une modification de la définition de U, qui pour rappel devient :

$$U = K_x K_y M_a + (K_x R_y + K_y R_x) - R_x R_y D + M_a P \equiv |X\,Y(-D)|_P \quad .$$

Pour que U soit toujours positif, nous avons rajouté un terme $M_a P$ car par définition $M_a P > R_x R_y D$. Cet ajout est reporté sur la borne sur U qui devient

$$U \leqslant \quad (C(\alpha,\mu) + M_a)\, P + 4M_a \quad .$$

Finalement, la condition suivante est obtenue :

$$M_b > \frac{1}{\alpha - 2}\,(1 + M_a + C(\alpha,\mu)) > \frac{1}{\alpha - 2}\left(\frac{4M_a + PM_a + C(\alpha,\mu)P}{P}\right) > \frac{U}{(\alpha-2)P} \quad .$$

Le passage de la condition H_1 à H_2 ne change par contre en rien la condition sur M_c.

3.1.4 Sélection des paramètres

Pour utiliser efficacement l'algorithme SPRR, nous avons besoin des conditions H_1 et H_2 pour que U puisse être représenté sur seulement $\frac{3n}{2}$ moduli (cf. section 3.1.3). Ainsi, on utilise MR sur une demi-base $\mathcal{B}_b$ et une base complète $\mathcal{B}_{a|c}$ (à la fin de l'algorithme SPRR) au lieu des 2 bases complètes que l'on a dans l'état de l'art [48, 52, 64]. Nous verrons dans la section 3.3.1 qu'il est possible de se passer des hypothèses H_1 et H_2 pour certains motifs de calcul très particuliers.

Pour satisfaire les hypothèses H_1 et H_2, deux stratégies peuvent être adoptées. Premièrement, si nous supposons que P est fixe, il nous faut alors trouver une base $\mathcal{B}_a$ satisfaisant H_1 ou H_2. Ainsi, nous pourrions utiliser notre algorithme pour tous les cryptosystèmes, avec les paramètres standards pour P, comme ceux pour ECC [91]. Malheureusement, nous n'avons, jusqu'à présent, pas trouvé de bonne méthode de sélection des moduli pour parvenir à satisfaire l'une des 2 hypothèses. Si nous choisissions de façon aléatoire $\mathcal{B}_a$ pour un P fixé, alors U aurait une taille de 2ℓ au lieu de $\frac{3\ell}{2}$.

L'autre stratégie est de considérer le problème dans l'autre sens, et de trouver un P qui satisfasse l'hypothèse H_1 ou H_2 à partir d'une base $\mathcal{B}_a$ donnée. Par exemple, dans le cas de H_2, on a $P = \frac{M_a D - 1}{\mu}$ avec μ petit. En pratique, on peut choisir $\mu = 1$, tester des valeurs aléatoires pour D (de taille $\ell/2$ bits) et tester si $M_a D - 1$ est un nombre premier. Cette stratégie fonctionne mais empêche l'utilisation des P des standards pour ECC. Bien sûr, on ne peut pas appliquer SPRR à RSA car sa sécurité est directement liée à la valeur du module RSA et à sa factorisation, on ne va donc pas utiliser ses propriétés pour accélérer les calculs. Pour le logarithme discret, le paramètre P est généré avec une certaine forme, dont H_2 est un sous-cas (cf. section 3.2.1).

Choisir de nouveaux P pour les courbes elliptiques n'est pas un problème. En effet, jusqu'à maintenant les attaques n'arrivent pas à exploiter les caractéristiques du corps de base, mais portent sur les propriétés de la courbe (notamment son cardinal). C'est pour cela que les standards du NIST [91] proposent des corps spécialement conçus pour être très efficaces en numération simple de position binaire (le binaire « classique »). Par exemple,

le corps standardisé $P_{521} = 2^{521} - 1$ permet des réductions modulaires rapides, comme expliqué dans la section 1.2.2. De la même façon, des paramètres P peuvent être générés pour être favorables à une utilisation du RNS. De même, nous verrons dans la section 3.2.1 que l'hypothèse H_2 peut être utilisée pour générer des paramètres pour les cryptosystèmes Elgamal et Diffie-Hellman.

Nous allons maintenant discuter de l'utilisation des 3 bases $\mathcal{B}_a$, $\mathcal{B}_b$ et $\mathcal{B}_c$. Tout d'abord, nous soulignons que $\mathcal{B}_a$ a un statut particulier dû à son implication dans les deux hypothèses H_1 et H_2. Ceci étant dit, les deux autres bases pourraient voir leurs rôles échangés dans une des étapes. En fait, si la base $\mathcal{B}_b$ est utilisée pour effectuer la deuxième extension de base de l'étape `Split` et la première de la réduction `MR` finale, c'est pour factoriser certains pré-calculs, réutilisés pour ces 2 opérations. On sauve ainsi $(n^2/4) + (n/2)$ mots élémentaires de w bits à stocker (`EMW` pour *elementary memory word*).

Dans le cadre d'une analyse plus fine des paramètres, on peut noter que la taille de M_b est plus grande de quelques bits que celle de M_a. Par exemple, pour $\mu = 1$ et $\alpha = 3$, la condition de la proposition 1 devient $M_b > 9D + 4M_a + 16$. Ces quelques bits supplémentaires peuvent être facilement obtenus, en prenant par exemple un ou deux éléments de $\mathcal{B}_a$ avec $w - 1$ bits et un ou deux de $\mathcal{B}_b$ avec $w + 1$ bits. Ce genre d'effet a été ici négligé pour les décomptes d'opérations. Nous avons supposé que les multiplications de w, $w + 1$ et $w - 1$ bits ont toutes le même coût. C'est notamment le cas lorsqu'on utilise les blocs multiplieurs dans les FPGA, qui ont une taille fixée, tant que $w + 1$ est plus petit ou égal à la taille limite supportée par un bloc multiplieur.

3.2 Applications

Les performances théoriques de notre méthode `SPRR` sont comparées ci-dessous à la multiplication de Montgomery RNS de l'état de l'art [48]. Nous allons analyser dans cette section le nombre de multiplications modulaires élémentaires de w bits (`EMM`) et le nombre de mots mémoire élémentaires (`EMW`). Le nombre d'`EMM` est la métrique utilisée pour effectuer les comparaisons dans l'état de l'art [12, 48, 95]. La plupart du temps en RNS, il y a, à peu près, autant de multiplications que d'additions élémentaires, car les parties les plus coûteuses du calcul modulaire en RNS sont les extensions de base. Ces extensions, que ce soit avec le CRT ou via le MRS, utilisent grossièrement le même nombre d'additions et de multiplications (succession de multiplications-additions). Les coûts ici seront donc exprimés en nombre de multiplications. Enfin, une dernière métrique globale sera proposée, le produit `EMM` $\times$ `EMW`. En RNS, le nombre de pré-calculs est important, c'est pourquoi cette métrique est prise en compte. Cette métrique permet, dans une certaine mesure, de donner une idée des améliorations que l'on peut espérer sur le compromis temps-surface pour une implantation matérielle du `SPRR`. On peut d'ailleurs noter que pour réduire le coût en surface d'une implantation RNS, on peut réduire le nombre de `Rower`s, mais par contre la quantité de pré-calculs à stocker, elle, ne change pas. La part de circuit allouée à la mémoire sera donc de plus en plus importante vis-à-vis de celle allouée aux `Rower`s lorsqu'on cherchera a obtenir une implantation compacte.

La table 3.1 présente le coût théorique de certaines opérations (en `EMM`), classiques en cryptographie asymétrique, pour quelques motifs d'opérations dans le corps. Pour ces opérations, nous considérons directement la représentation de Montgomery en RNS afin de simplifier les notations, autrement dit $|AB|_P$ est en fait $|ABM_aM_b|_P$. Nous verrons ensuite

Opérations	$\|AB\|_P$	$\|A^2\|_P$	$\|Cst \times A\|_P$
MM [EMM]	$2n^2 + 4n$	$2n^2 + 4n$	$2n^2 + 4n$
SPRR [EMM]	$2.5n^2 + 12.5n$	$1.75n^2 + 10.5n$	$1.75n^2 + 7n$

TABLE 3.1 – Comparaison du coût théorique d'opérations courantes en cryptographie, en nombre de multiplications élémentaires EMM (Cst est une constante).

Split	PR	MR
$\overrightarrow{\left(T_a^{-1}\right)_a}$: $n/2$	$\overrightarrow{D_{a\|b\|c}}$: $3n/2$	$\overrightarrow{\left(-P^{-1}T_b^{-1}\right)_b}$: $n/2$
$\overrightarrow{(T_{a,i})_{b\|c}}$: $n^2/2$	$\overrightarrow{P_{a\|b\|c}}$: $3n/2$	$\overrightarrow{(T_{b,i})_c}$: $n^2/4$
$\overrightarrow{(M_a)_{b\|c}}$: n		$\overrightarrow{(-M_b)_c}$: $n/2$
$\overrightarrow{\left(M_a^{-1}\right)_c}$: $n/2$		$\overrightarrow{\left(M_b^{-1}T_{a\|c}^{-1}\right)_{a\|c}}$: n
$\overrightarrow{\left(M_a^{-1}T_b^{-1}\right)_b}$: $n/2$		$\overrightarrow{\left(T_{a\|c,i}\right)_b}$: $n^2/2$
$\overrightarrow{(T_{b,i})_a}$: $n^2/4$		$\overrightarrow{\left(M_{a\|c}\right)_b}$: $n/2$
$\overrightarrow{(-M_b)_a}$: $n/2$		
Total : $3n^2/2 + 17n/2$		

TABLE 3.2 – Décompte du nombre de pré-calculs à stocker en mots de w bits (EMWs) pour notre algorithme SPRR.

que grâce aux factorisations (notamment des Splits) et à la réutilisation d'opérandes, le coût total peut être bien plus réduit dans une séquence d'opérations que dans la somme des opérations individuelles. Dans la table 3.1, nous voyons notamment que la multiplication modulaire coûte moins cher avec l'algorithme de l'état de l'art dans le cas de 2 opérandes distinctes, c'est-à-dire $|AB|_P$. Par contre, le carré modulaire et la multiplication par une constante coûtent moins cher (du moins asymptotiquement) que l'algorithme de l'état de l'art, et c'est là l'intérêt de la solution proposée.

Pour parvenir à ce résultat, il nous faut d'abord compter pour chacune des opérations le nombre d'étapes Split requises, puis rajouter le coût de l'étape PR en utilisant les optimisations proposées dans la section 3.1 et enfin rajouter le coût d'une réduction finale MR sur bases réduites. Pour $|AB|_P$, on a besoin de deux Split, de l'étape PR sans optimisation particulière (autre que l'utilisation de Karatsuba) et enfin d'une réduction MR, ce qui donne $2 \times \left(\frac{3}{4}n^2 + \frac{3}{2}n\right) + (7n) + \left(n^2 + \frac{5}{2}n\right) = \frac{5}{2}n^2 + \frac{25}{2}n = 2.5n^2 + 12.5n$ EMM. Le carré $|A^2|_P$ ne nécessite lui qu'un seul Split. De plus, on peut gagner un peu sur l'étape PR en économisant certains produits, ce qui nous donne $\left(\frac{3}{4}n^2 + \frac{3}{2}n\right) + (6.5n) + \left(n^2 + \frac{5}{2}n\right) = 1.75n^2 + 10.5n$ EMM. Enfin, en multipliant par une constante, on réduit le coût de PR à $3n$ au lieu de $6.5n$ pour le carré, ce qui donne $1.75n + 7n$.

La table 3.2 présente le décompte des valeurs à pré-calculer pour notre algorithme,

étape par étape, en mots mémoire élémentaires EMW. Grâce à la réduction du nombre de moduli de $2n$ à $\frac{3}{2}n$ et à l'utilisation d'extensions de base uniquement sur des bases réduites, le nombre d'EMW est réduit. En effet, dans l'article de Gandino *et al.* [48], le nombre de pré-calculs s'élève à $2n^2 + 10n$ contre $\frac{3}{2}n^2 + \frac{17}{2}n$ pour notre algorithme SPRR. La figure 3.2 illustre le rapport SPRR / MM en terme de EMW. On peut observer jusqu'à 25 % de réduction du nombre d'EMM, avec au moins 20% de réduction dès que n est supérieur à 5.

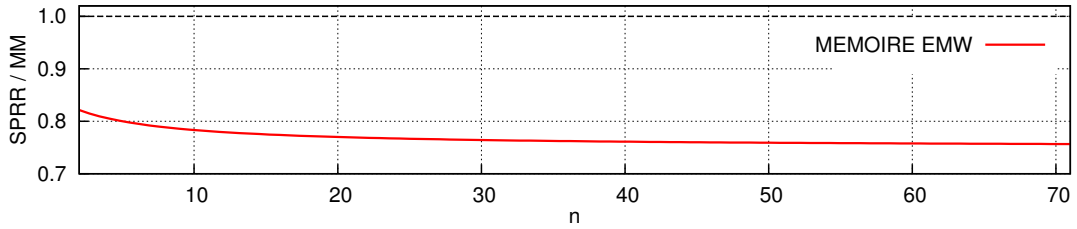

FIGURE 3.2 – Rapport du nombre d'EMW pré-calculés de notre solution SPRR vis à vis de l'état de l'art MM, pour des nombres de moduli classiques en cryptographie asymétrique.

3.2.1 Application au logarithme discret

Nous allons ici étudier l'application de l'algorithme SPRR au cas des exponentiations dans $\mathbb{F}_P$ pour des cryptosystèmes comme Diffie-Hellman [38] et Elgamal [44].

Comme décrit dans le chapitre 1, pour ces cryptosystèmes basés sur le logarithme discret, on choisit un P premier de 1024 à 3072 bits avec $P-1 = VQ$ et Q un premier de 160 à 256 bits. En fait, cette condition correspond à l'hypothèse H_2, avec $\mu = 1$ et D un multiple de Q : $P - 1 = VQ = M_a V'Q = M_a D$. On peut alors générer P sous cette hypothèse, et l'utiliser pour les calculs de logarithme discret. On calcule ensuite des exponentiations de G où G est un générateur du sous-groupe d'ordre Q de $\mathbb{F}_p^*$.

Une première application pour le SPRR est l'algorithme 22, la très utilisée « échelle de Montgomery » (voir [62]), utilisé comme protection contre certaines attaques par canaux cachés. Pour chacun des bits de l'exposant, deux multiplications sont effectuées. On remarque que pour ces 2 multiplications modulaires, on a seulement 2 opérandes différentes, R_0 et R_1, ce qui implique seulement 2 étapes Split. Dans ce motif de calcul, notre algorithme va donc être efficace, avec un coût de $3.5n^2 + 19n$ EMMs contre $4n^2 + 8n$ EMMs pour MM. Si on prend aussi en compte la mémoire pour le coût global, nous obtenons $5.25n^4 + 58.25n^3 + 161.5n^2$ EMM × EMW pour SPRR contre $8n^4 + 56n^3 + 80n^2$ EMM × EMW pour MM.

La figure 3.3 illustre ces résultats théoriques pour l'échelle de Montgomery, mais aussi pour l'algorithme LSBF 5 présenté au chapitre 1 avec une notation additive de la loi de groupe. Dans cet algorithme, si le bit d'exposant vaut 1, on effectue le même motif d'opérations que dans le cas de l'échelle de Montgomery. Lorsque ce bit vaut 0, seul un carré est effectué, ce qui reste un bon motif d'opérations pour effectuer notre algorithme. Les résultats entre les deux exponentiations sont très similaires comme le montre la figure 3.3, que ce soit en terme de nombre d'opérations EMM ou en coût global EMM × EMW. Les tailles habituelles pour ces exponentiations sont 1024 et 2048 bits, et correspondent à $n = 33$ et $n = 66$ pour des mots de $w = 32$ bits (voir la table 1.6). Pour $n = 33$ et $n = 66$, on obtient respectivement une réduction de 4 % et 9 % du nombre d'EMM et jusqu'à 30 % de réduction sur le coût total EMM × EMW.

Algorithme 22: Exponentiation « échelle de Montgomery » [62].

Entrées : G, $E = (e_{t-1}, \ldots, e_0)_2$
Sortie : $S = G^E \bmod P$

$R_0 \leftarrow 1$; $R_1 \leftarrow G$
pour de $j = t-1$ **à** 0 **faire**
  si $e_j = 0$ **alors**
    $R_1 \leftarrow R_0 R_1$; $R_0 \leftarrow R_0^2$
  sinon
    $R_0 \leftarrow R_0 R_1$; $R_1 \leftarrow R_1^2$
retourner R_0

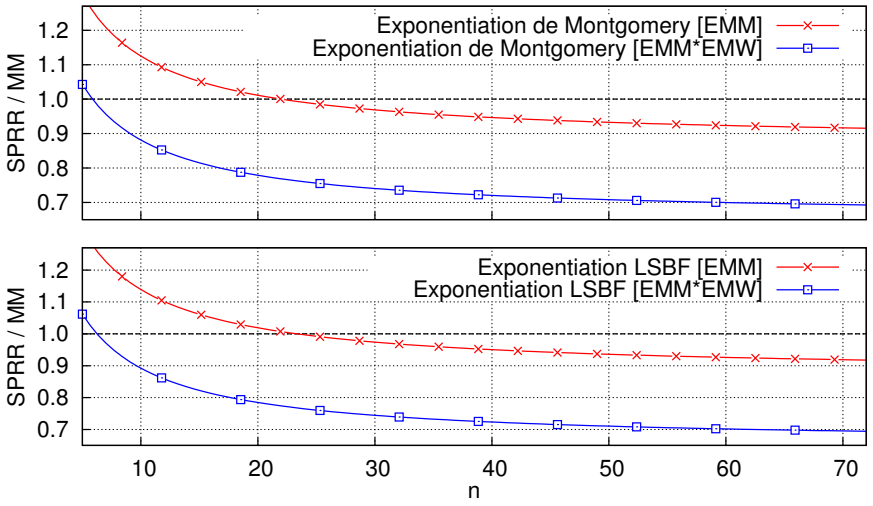

FIGURE 3.3 – Comparaison théorique des performances (`EMM`) et du coût (`EMM` × `EMW`) des algorithmes `SPRR` et `MM` pour les algorithmes d'exponentiations 22 (échelle de Montgomery) et 5 (LSBF, poids faibles en tête).

3.2.2 Applications aux courbes elliptiques

Nous allons maintenant présenter des exemples d'applications de notre algorithme sur des formules d'opérations sur les points d'une courbe elliptique, comme l'addition et le doublement. Nous présentons ici trois ensembles de formules. Afin d'évaluer les différents coûts, nous avons procédé de la même façon que pour le logarithme discret. Nous n'allons pas ici détailler ces calculs car, d'une part, dans le prochain chapitre nous présentons un bien meilleur algorithme de multiplication applicable uniquement pour les courbes elliptiques et d'autre part, la complexité des formules rend ce décompte fastidieux. Ces résultats sont présentés car ils montrent tout de même comment se comporte notre algorithme sur diverses applications d'additions ou de doublements de points. Pour le décompte des opérations en utilisant l'algorithme `SPRR`, nous avons d'abord compté le nombre d'éléments à décomposer en utilisant `Split`, puis le nombre d'appels à `PR` en adaptant le coût suivant que l'on effectue une multiplication, un carré ou un produit par une constante. Enfin, le nombre de `MR` sur base réduite est exactement le nombre de réductions `MR` de l'état de l'art. Ces décomptes mènent finalement aux résultats présentés dans les tables 3.3, 3.4 et 3.5. Ces tables fournissent des complexités dépendantes de n et sont illustrées graphiquement dans les figures 3.4 et 3.5.

Le premier ensemble de formules est présenté dans la table 3.3. Ce sont les formules issues de [17] qui sont applicables pour le cas classique des courbes représentées par l'équation de Weierstrass courte en coordonnées Jacobiennes. On rappelle que les coordonnées Jacobiennes sont définies par (X, Y, Z) avec $x = X/Z^2$ et $y = Y/Z^3$. D'après [17], ce sont celles qui requièrent le moins d'opérations pour ces courbes là avec ces coordonnées. Cette table propose les coûts pour 3 opérations de points : le doublement, le triplement et l'addition mixte. L'addition mixte est une addition où l'un des deux points en entrée est représenté en coordonnées affines (c.-à-d. $Z = 1$), elle est donc moins coûteuse qu'une addition avec deux points quelconques. Lorsqu'on utilise l'algorithme 5 pour effectuer la multiplication scalaire, alors le point $\mathbf{P}$, fixe, va être représenté en coordonnées affines afin de pouvoir faire une addition mixte. Ces formules d'addition mixte sont d'autant plus intéressantes avec le `SPRR` car elles permettent plus de réutilisation qu'une addition de points quelconques. On fournit aussi les résultats pour le doublement de point et le triplement, utilisé pour accélérer la multiplication scalaire en utilisant le recodage de clé DBNS, voir par exemple [40]. On remarque que pour ces formules, notre algorithme est toujours asymptotiquement meilleur que `MM`, que ce soit en nombre d'opérations `EMM` ou en coût global `EMM` $\times$ `EMW`. Par contre, les constantes devant n étant plus grandes, nous verrons que ces formules ne sont meilleures que pour de hauts niveaux de sécurité.

Un autre exemple de formules est fourni dans la table 3.4, issu de [8] et utilisé dans l'implantation ECC de l'état de l'art en RNS [52]. Ces formules ont été spécialement adaptées pour le RNS et pour l'utilisation de l'échelle de Montgomery en tant que contre-mesure. Ces formules ont été proposées afin de réduire le nombre de réductions modulaires RNS, notamment via l'utilisation de motifs de type « sommes de produits » avant de procéder à la réduction modulaire. Ces formules ont la particularité de proposer un doublement de point plus cher qu'une addition en RNS avec l'algorithme de l'état de l'art `MM`, comme le montre la table 3.4. Pour ces formules, on observe que l'addition de points est toujours meilleure avec l'algorithme de l'état de l'art `MM`, alors que le doublement est meilleur avec `SPRR`, toujours grâce à un plus haut taux de réutilisation. En tenant compte du fait que pour chaque bit de clé, on effectue une addition et un doublement dans l'algorithme échelle de Montgomery, on a alors une meilleure complexité asymptotique pour notre algorithme `SPRR`.

	Formules	MM	SPRR
$\mathbf{P_1 + P_2}$ (mADD)	$A_1 = Z_1^2$ $U_2 = X_2 A_1$ $S_2 = Y_2 Z_1 A_1$ $H = U_2 - X_1$ $H_2 = H^2$ $I = 4H_2$ $J = HI$ $R = 2(S_2 - Y_1)$ $V = X_1 I$ $X_3 = R^2 - J - 2V$ $Y_3 = R(V - X_3) - 2Y_1 J$ $Z_3 = (Z_1 + H)^2 - A_1 - H_2$	EMM : $20\,n^2 + 50n$ EMM × EMW : $40\,n^4 + 300\,n^3 + 500\,n^2$	EMM : $17.5\,n^2 + 95\,n$ EMM × EMW : $26.25\,n^4 + 291.25\,n^3 + 807.5\,n^2$
$2\ \mathbf{P_1}$	$A = X_1^2$ $B = Y_1^2$ $C = B^2$ $D = Z_1^2$ $S = 2\left((X_1 + B)^2 - A - C\right)$ $M = 3A + aD^2$ $T = M^2 - 2S$ $X_3 = T$ $Y_3 = M(S - T) - 8C$ $Z_3 = (Y_1 + Z_1)^2 - B - D$	EMM : $20\,n^2 + 48\,n$ EMM × EMW : $40\,n^4 + 296\,n^3 + 480\,n^2$	EMM : $16\,n^2 + 100.5\,n$ EMM × EMW : $24\,n^4 + 286.75\,n^3 + 854.25\,n^2$
$3\ \mathbf{P_1}$	$A = X_1^2$ $B = Y_1^2$ $C = Z_1^2$ $D = B^2$ $M = 3A + aC^2$ $N = M^2$ $E = 6\left((X_1 + B)^2 - A - D\right) - N$ $F = E^2$ $T = 16D$ $U = (M + E)^2 - N - F - T$ $X_3 = 4(X_1 F - 4BU)$ $Y_3 = 8Y_1\left(U(T - U) - EF\right)$ $Z_3 = (Z_1 + E)^2 - C - F$	EMM : $28\,n^2 + 72\,n$ EMM × EMW : $56\,n^4 + 424\,n^3 + 720\,n^2$	EMM : $23\,n^2 + 160\,n$ EMM × EMW : $34.5\,n^4 + 435.5\,n^3 + 1360\,n^2$

TABLE 3.3 – Coûts en multiplications EMM et coûts globaux EMM × EMW des algorithmes MM et SPRR pour les formules efficaces issues de [17] pour courbes sous forme de Weierstrass courte ($y^2 = x^3 + ax + b$, avec a quelconque), en coordonnées Jacobiennes.

	Formules	MM	SPRR
$\mathbf{P_1 + P_2}$	$A = Z_1X_2 + Z_2X_1$ $B = 2X_1X_2$ $C = 2Z_1Z_2$ $D = aA + bC$ $Z_3 = A^2 - BC$ $X_3 = BA + CD + 2X_GZ_3$	EMM : $12n^2 + 34n$ EMM × EMW : $24n^4 + 168n^3 + 340n^2$	EMM : $12.75n^2 + 67n$ EMM × EMW : $19.125n^4 + 209.375n^3 + 569.5n^2$
$2\ \mathbf{P_1}$	$E = Z_1^2$ $F = 2X_1Z_1$ $G = X_1^2$ $H = -4bE$ $I = aE$ $X_3 = FH + (G - I)^2$ $Z_3 = 2F(G + I) - EH$	EMM : $14n^2 + 32n$ EMM × EMW : $28n^4 + 204n^3 + 320n^2$	EMM : $12.25n^2 + 75n$ EMM × EMW : $18.375n^4 + 216.625n^3 + 637.5n^2$

TABLE 3.4 – Coûts en multiplications EMM et coûts globaux EMM × EMW des algorithmes MM et SPRR pour les formules optimisées RNS issues de [8] pour les courbes sous forme de Weierstrass courte ($y^2 = x^3 + ax + b$, avec a quelconque), en coordonnées (X, Z) et adaptées à l'échelle de Montgomery.

Le troisième ensemble de formules concerne les opérations de points sur les courbes d'Edwards [43], qui sont de la forme $x^2 + y^2 = c^2(1 + dx^2y^2)$ avec $d \notin \{0, 1\}$. Ces formules utilisent les coordonnées inversées proposées dans [16], où les points sont représentés par $(Z/X, Z/Y)$. Les formules présentées dans la table 3.5 sont une amélioration de celles disponibles dans [16], et présentées dans [17] comme les moins coûteuses pour ces courbes en nombre de multiplications modulaires. On constate dans ce cas aussi que le gain en terme de complexité dépend du niveau de réutilisation que les formules permettent, les gains se situent donc principalement dans les doublements et triplements.

Pour finir, les figures 3.4 et 3.5 résument les gains théoriques que l'on obtient pour nos différents ensembles de formules. Dans ces figures, ADD représente l'addition, mADD l'addition mixte, DBL le doublement et TPL le triplement de points.

La figure 3.4 détaille les résultats pour les formules utilisant les coordonnées Jacobiennes pour la forme de Weierstrass courte. La courbe inférieure représente exactement les résultats de complexité de la table 3.3, avec le coût en EMM de chacune des opérations de points. On remarque que le doublement est meilleur avec le SPRR à partir de $n = 13$ moduli, alors que le doublement et l'addition mixte ne le sont qu'à partir de 18. Les courbes présentées dans la partie supérieure de la figure 3.4 combinent ces coûts pour refléter le gain sur une multiplication scalaire. Le motif 2DBL+mADD correspond ainsi à un algorithme de doublement et addition, le motif 2DBL+mADD+TPL correspond lui à un algorithme de multiplication scalaire utilisant le recodage DBNS du scalaire (voir [40]). Ce dernier motif surestime un peu le nombre d'additions par rapport au nombre de doublements et triplements, afin de simplifier l'expression (ce qui désavantage un peu notre algorithme). Le coût en EMM des deux motifs sont très proches. On remarque que notre algorithme est meilleur que MM pour $n \geqslant 16$, en nombre d'opérations effectuées. C'est le cas si les calculs sont effectués sur un corps de 521 bits avec $w = 32$. Sur une plateforme avec une contrainte sur la taille des moduli, comme dans [5] où $w = 16$ pour du calcul sur GPU. Alors les courbes sur 384 et 521 bits correspondent respectivement à $n = 24$ et $n = 34$, où notre algorithme

	Formules	MM	SPRR
$\mathbf{P_1} + \mathbf{P_2}$ (mADD)	$A = Z_1 Z_2$ $B = dA^2$ $C = X_1 X_2$ $D = Y_1 Y_2$ $E = CD$ $H = C - D$ $I = (X_1 + Y_1) - C - D$ $X_3 = c(E + B)H$ $Y_3 = c(E - B)I$ $Z_3 = AHI$	EMM : $26n^2 + 60n$ EMM × EMW : $52n^4 + 380n^3 + 600n^2$	EMM : $25.75n^2 + 131.5n$ EMM × EMW : $38.625n^4 + 416.125n^3 + 1117.75n^2$
$2\,\mathbf{P_1}$	$A = X_1^2$ $B = Y_1^2$ $C = A + B$ $D = A - B$ $E = \left((X_1 + Y_1)^2 - C\right)$ $X_3 = CD$ $Y_3 = E(C - u_2 Z_1^2)\star$ $Z_3 = cDE$	EMM : $18n^2 + 36n$ EMM × EMW : $36n^4 + 252n^3 + 360n^2$	EMM : $15.75n^2 + 86.5n$ EMM × EMW : $23.625n^4 + 263.625n^3 + 367.625n^2$
$3\,\mathbf{P_1}$	$A = X_1^2$ $B = Y_1^2$ $C = Z_1^2$ $D = A + B$ $F = D^2$ $E = 4(D - u_1\,C)\star$ $H = 2D(B - A)$ $S = F - AE$ $Q = F - BE$ $T = Q^2$ $X_3 = (H + Q)\left((Q + X_1)^2 - T - A\right)$ $Y_3 = 2(H - S)SY_1$ $Z_3 = S\left((Q + Z_1)^2 - T - C\right)$	EMM : $30n^2 + 60n$ EMM × EMW : $60n^4 + 420n^3 + 600n^2$	EMM : $24.75n^2 + 144.5n$ EMM × EMW : $37.125n^4 + 402.125n^3 + 614.125n^2$

TABLE 3.5 – Coûts en multiplications EMM et coûts globaux EMM×EMW pour les algorithmes MM et SPRR des formules issues de [17] pour courbes d'Edwards, avec les coordonnées inversées. Note $\star$: $u_1 = c^2 d$ and $u_2 = 2\,u_1$ pour une courbe définie par $x^2 + y^2 = c^2(1 + dx^2y^2)$.

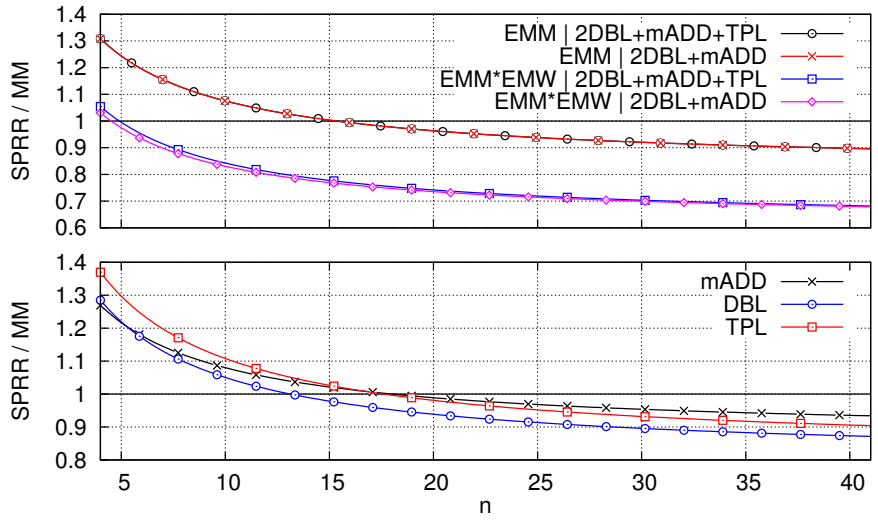

FIGURE 3.4 – Comparaison des performances et des coûts théoriques entre les algorithmes `SPRR` et `MM` pour les formules présentées à la table 3.3 (en bas) et pour des combinaisons classiques présentes dans les algorithmes de multiplication scalaire.

est meilleur que `MM` de 4.5% et 9.5% respectivement. La figure supérieure propose aussi une métrique du coût global, avec le produit opérations `EMM`/mémoire `EMW` pour les deux motifs. Suivant cette métrique `EMM` × `EMW`, notre algorithme est moins coûteux que l'algorithme de l'état de l'art, même pour des petites valeurs de $n \geqslant 5$. Ainsi, pour une base $n \geqslant 16$, on obtient avec `SPRR` une réduction de plus de 25 % du coût global par rapport à `MM`.

La figure 3.5 présente les résultats pour les deux autres ensembles de courbes. Ces résultats sont clairement moins bons que ceux obtenus avec les formules utilisant l'addition mixte. Sur la partie inférieure de la figure 3.5 sont présentés les résultats pour l'échelle de Montgomery et les formules de la table 3.4. Le coût global est réduit à partir de $n = 9$ mais le nombre d'opérations est toujours plus grand que pour l'algorithme de l'état de l'art. De même pour la partie supérieure représentant les calculs sur les courbes d'Edwards. Le coût global est cette fois-ci meilleur très rapidement, lorsque $n \geqslant 5$, mais le nombre d'opérations est tout juste équivalent lorsqu'on atteint $n = 26$. Ces courbes montrent clairement que le coût de l'algorithme `SPRR` dépend grandement de la séquence de calcul sur laquelle il est appliqué et des réutilisations qui sont faites. De plus, les gains étant sur des complexités asymptotiques, ils sont bien plus clairs et importants pour les algorithmes d'exponentiation que pour la cryptographie sur courbes elliptiques. Nous verrons dans le chapitre 4 un algorithme qui ne sera pas applicable pour les exponentiations mais bien plus adapté aux calculs sur courbes elliptiques.

3.3 Exponentiation rapide RNS sans hypothèse sur P

Cette section va présenter de nouveaux algorithmes d'exponentiation rapide pour le RNS [1], réduisant le nombre d'opérations par rapport à l'algorithme de l'état de l'art [48]. Ces algorithmes reprennent les idées qui sont à la base de la multiplication `SPRR` afin de profiter des réutilisations des opérandes dans certains algorithmes d'exponentiation. Si ces algorithmes sont présentés dans une section à part, c'est qu'ils n'utilisent pas directement

1. Ces résultats ne sont pas dans le papier d'ASAP 2014 [20]

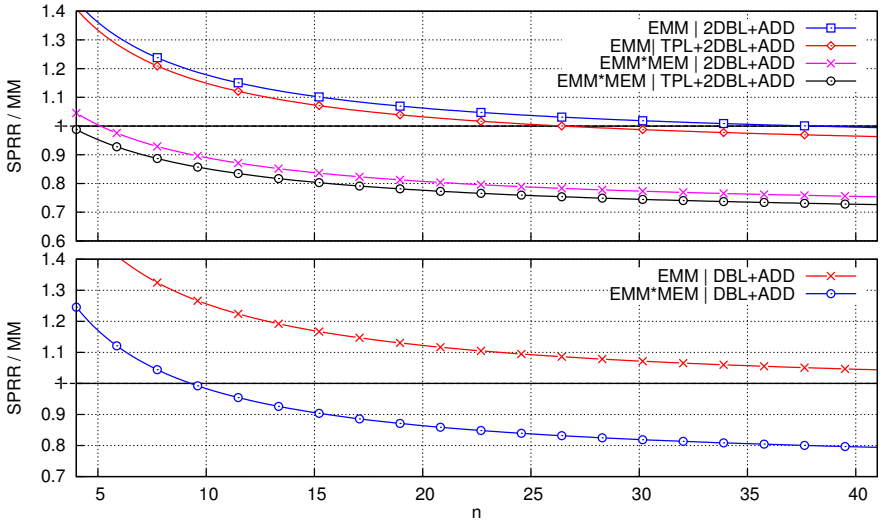

FIGURE 3.5 – Comparaison des performances et des coûts théoriques entre les algorithmes `SPRR` et `MM` pour des combinaisons classiques des formules présentées à la table 3.4 (en bas) et table 3.5 présentes dans les algorithmes de multiplication scalaire.

le `SPRR` à la différence des exponentiations étudiées dans la section 3.2.1. En effet, ces nouveaux algorithmes vont accélérer les exponentiations sans aucune hypothèse sur P et réduisent encore le nombre de multiplications élémentaires `EMM` par rapport à l'utilisation de `SPRR`. En contrepartie, le nombre de moduli est $2n$ (comme pour l'algorithme de l'état de l'art). De plus, ils augmentent le nombre de pré-calculs à stocker par rapport au `SPRR` et à l'algorithme de l'état de l'art [48]. Enfin, ces algorithmes sont basés sur un motif d'opération particulier, certains algorithmes, comme par exemple l'échelle de Montgomery, ne calculent pas ce motif et ne sont donc pas accélérés par rapport au `SPRR`.

3.3.1 Un nouvel algorithme d'exponentiation RNS

Algorithme 23: Exponentiation carré et multiplication (source [51]).

Entrées : $k = (k_{\ell-1}, \ldots, k_1, k_0)_2$, $G \in \mathbb{Z}/P\mathbb{Z}$
Sortie : $G^k \bmod P$

1 $S \leftarrow 1$
2 **pour** i **de** $\ell - 1$ **à** 0 **faire**
3 $\quad S \leftarrow S^2 \bmod P$
4 $\quad$ **si** $k_i = 1$ **alors** $S \leftarrow S \times G$
5 **retourner** S

Le calcul que nous allons étudier et accélérer dans cette partie est $X^2C \bmod P$, où C est une constante pré-calculée. On retrouve ce motif dans l'algorithme 23, qui est l'exponentiation équivalente à la multiplication scalaire de l'algorithme 6 (présenté dans la section 1.1.4). L'algorithme 23 présenté suppose que l'on soit déjà en représentation de Montgomery RNS (ou dans la version améliorée de Gandino *et al.* [48]) et retourne le résultat dans cette même représentation. Ce n'est pas sur ces conversions que portent les améliorations, et ces conversions initiales et finales sont négligeables par rapport au coût de l'exponentiation. Avant d'analyser notre algorithme, nous allons d'abord expliquer l'idée,

proche du SPRR, qui lui sert de base.

On remarque que lorsque $k_i = 1$ dans l'algorithme 23, on effectue la séquence de calcul $S \leftarrow S^2 \bmod P$ puis $S \leftarrow S \times G \bmod P$, c'est-à-dire $S^2G \bmod P$ avec G constant. Pour accélérer ce calcul, nous allons décomposer S en (K_s, R_s) comme pour le SPRR. Pour ce faire, on va utiliser à nouveau l'algorithme 20 Split, toujours avec $n_a = n_b = n/2$ mais par contre avec $n_c = n$. Nous avons bien, comme annoncé, $n_a + n_b + n_c = 2n$ moduli. On rappelle que le coût de cet algorithme est $(n_a + n_a(n_b + n_c)) + (n_b + n_b n_a) + n_c$ EMM, c'est-à-dire $\left(\frac{n}{2} + \frac{3n^2}{4}\right) + \left(\frac{n}{2} + \frac{n^2}{4}\right) + n = n^2 + 2n$ EMM dans notre cas. Après avoir obtenu (K_s, R_s), on remarque que :

$$\begin{aligned} S^2G &\equiv \left(K_s^2 M_a^2 + 2K_s R_s M_a + R_s^2\right) G \bmod P \\ &\equiv K_s^2 |M_a^2 G|_P + K_s R_s |2M_a G|_P + R_s^2 |G|_P \bmod P \\ &\equiv K_s \left(K_s |M_a^2 G|_P + R_s |2M_a G|_P\right) + R_s^2 |G|_P \bmod P \quad . \end{aligned} \tag{3.7}$$

Si on pose $U_2 = K_s \left(K_s |M_a^2 G|_P + R_s |2M_a G|_P\right) + R_s^2 |G|_P$, alors $\log_2 U_2 \approx 2\ell$. Comme cela sera montré plus loin dans cette section, on peut borner U_2 plus précisément par :

$$U_2 < 12P^2 + M_a^2 P \quad , \tag{3.8}$$

en supposant que $S < 3P$. En calculant U_2, nous avons obtenu une valeur telle que $U_2 \equiv S^2G \bmod P$ de taille $2\ell + \varepsilon$ bits. Il suffit alors de choisir la base $\mathcal{B}_c$ pour pouvoir effectuer une réduction modulaire de Montgomory RNS MR prenant en entrée cette valeur de $2\ell + \varepsilon$ et de réduire.

Prenons l'exemple d'une exponentiation pour RSA 2048 bits, avec $\log_2 P = 2048$ (bien sûr ici P n'est pas premier). Supposons que S est représenté dans le domaine de Montgomery, avec $S < 3P$. Dans cet exemple, la valeur à réduire S^2G est de taille 6148 bits, le but est d'obtenir un résultat de 2050 bits (car inférieur à $3P$). Supposons que $n = 66$ et $w = 32$, alors M_a est un nombre de $33 \cdot 32 = 1056$ bits. En appliquant la borne de l'équation 3.8, on obtient $\log_2 U_2 < 4161$. En effet, $\log_2(M_a^2 P) = 2\log_2(M_a) + \log_2(P) = 2112 + 2048 = 4160$ et $\log_2(12P^2) = 4 + 2\log_2 P = 4100$. Pour résumer, on obtient U_2 un peu plus grand que P^2, en ayant seulement décomposé S puis calculé l'équation 3.7. Nous avons donc déjà parcouru la moitié du chemin pour passer de 6144 à 2049 bits, pour le coût du Split et de 5 multiplications RNS sur $2n$ moduli pour évaluer U_2. On obtient $n^2 + 2n + 5 \cdot 2 \cdot n = n^2 + 12n$ EMM pour cette première partie. Ensuite, on applique la réduction MR de l'état de l'art qui coûte $2n^2 + 2n$ EMM. Dans notre algorithme 24, les constantes pré-calculées pour le calcul de U_2 sur la base $\mathcal{B}_c$ permettent d'inclure directement le calcul de la ligne 1 de l'algorithme 15 MR ainsi que la ligne 1 de l'algorithme 14 BE, réduisant le coût du MR final à $2n^2 + n$. Le coût de l'opération $S^2G \bmod P$ est donc de $3n^2 + 13n$ EMM, contre $4n^2 + 8n$ EMM pour l'état de l'art. Dans l'état de l'art, on effectue en fait 2 multiplications sur les $2n$ moduli suivies de 2 MR d'où $2(2n^2 + 2n) + 2n$ EMM. Pour notre exemple avec $n = 66$, on obtient 13926 EMM au lieu de 17952 EMM pour effectuer le carré et la multiplication lorsque $k_i = 1$ dans l'algorithme, soit environ une réduction de 22.5 %. Puisqu'on a en moyenne autant de 0 que de 1 dans k_i, le rapport des complexités en EMM pour l'exponentiation complète entre notre algorithme et l'état de l'art est $\frac{2n^2+4n+3n^2+13n}{3(2n^2+4n)} = \frac{5n^2+17n}{6n^2+12n}$. Pour $n = 66$, on a une réduction du nombre d'EMM de 15.0 % et pour $n = 34$ (pour RSA-1024 par exemple) on a un gain de 13.4 %. La figure 3.6 illustre ce ratio pour $n > 5$. On remarque que la convergence vers $\frac{1}{6}$ est rapide, on atteint un réduction de 10 % dès $n > 15$.

Algorithme 24: Exponentiation RNS revisitée

Entrées : $k = (k_{\ell-1}, \ldots, k_1, k_0)_2$

Pré-calculs : $\overrightarrow{\left(M_{c,i}^{-1}P^{-1}|GM_a^2|_P\right)_c}$, $\overrightarrow{\left(M_{c,i}^{-1}P^{-1}|2GM_a|_P\right)_c}$, $\overrightarrow{\left(M_{c,i}^{-1}P^{-1}G\right)_c}$ dans $\mathcal{B}_c$

Pré-calculs : $\overrightarrow{\left(|GM_a^2|_P\right)_{a|b}}$, $\overrightarrow{\left(|2GM_a|_P\right)_{a|b}}$, $\overrightarrow{G_{a|b}}$ dans $\mathcal{B}_{a|b}$

Sortie : $\overrightarrow{|G^k|_P}$ dans $\mathcal{B}_{a|b|c}$

$\overrightarrow{S_{a|b|c}} \leftarrow \overrightarrow{1}$

pour i *de* $\ell - 1$ *à* 0 **faire**

  si $k_i = 0$ **alors**

    $\overrightarrow{S_{a|b|c}} \leftarrow \overrightarrow{S_{a|b|c}} \times \overrightarrow{S_{a|b|c}}$

    $\overrightarrow{S_{a|b|c}} \leftarrow \texttt{MR}(\overrightarrow{S_c}, \overrightarrow{S_{a|b}})$

  sinon

    $(\overrightarrow{K_s}, \overrightarrow{R_s}) \leftarrow \texttt{Split}(\overrightarrow{S})$

    $\overrightarrow{S_{a|b}} \leftarrow \overrightarrow{(K_s)_{a|b}} \times \left(\overrightarrow{(K_s)_{a|b}} \times \overrightarrow{\left(|GM_a^2|_P\right)_{a|b}} + \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{(|2GM_a|_P)_{a|b}}\right)$

    $\overrightarrow{S_{a|b}} \leftarrow \overrightarrow{S_{a|b}} + \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{G_{a|b}}$

    $\overrightarrow{S_c} \leftarrow \overrightarrow{(K_s)_c} \times \overrightarrow{\left(M_{c,i}^{-1}P^{-1}|GM_a^2|_P\right)_c} + \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{\left(M_{c,i}^{-1}P^{-1}|2GM_a|_P\right)_c}$

    $\overrightarrow{S_c} \leftarrow \overrightarrow{(K_s)_c} \times \overrightarrow{S_c} + \overrightarrow{(R_s)_c} \times \overrightarrow{(R_s)_c} \times \overrightarrow{\left(M_{c,i}^{-1}P^{-1}G\right)_c}$

    $\overrightarrow{S_{a|b|c}} \leftarrow \texttt{MR}(\overrightarrow{S_c}, \overrightarrow{S_{a|b}})$

retourner $\overrightarrow{S_{a|b|c}}$

La proposition ci-dessous résume les conditions pour lesquelles l'algorithme 24 retournera un résultat exact.

Proposition 3.
Soient $\mathcal{B}_a$, $\mathcal{B}_b$ et $\mathcal{B}_c$ trois bases RNS telles que $M_aM_b > 3P$, avec $\lfloor \log_2 M_a \rfloor = \lfloor \log_2 M_b \rfloor$ et $M_c > 12P + M_a^2$. Alors la sortie $\overrightarrow{S_{a|b|c}}$ de l'algorithme 24 est telle que $S_{a|b|c} < 3P$ et $S_{a|b|c} \equiv G^k \bmod P$.

Si nous revenons à l'exemple RSA 2048 bits avec $n = 66$ et $w = 32$, nous pouvons remarquer que nous avons déjà $M_aM_b > 3P$, puisque M_aM_b est un entier de 2112 bits, contre 2050 pour $3P$. Ensuite, la condition $M_c > 12P + M_a^2$ implique, entre autres, que $\log_2 M_c \geqslant 2113$, on aura donc au moins un moduli de $\mathcal{B}_c$ avec un bit supplémentaire ($2113 = 66 \cdot 32 + 1$).

Pour conclure, on peut noter que la modification effectuée pour le cas $k_i = 1$ n'altère pas la représentation de Montgomery. En effet, lorsque $k_i = 1$, nous calculons la réduction modulaire MR sur la valeur U_2, avec $U_2 \equiv S^2R^2G \bmod P$, le résultat est donc bien $S^2RG \bmod P$.

Analyse détaillée de l'algorithme

Nous allons ci-dessous montrer la borne présentée précédemment dans l'équation 3.8, et en déduire les conditions de la proposition 3. Ensuite nous discuterons du surcoût en terme de pré-calculs nécessaires.

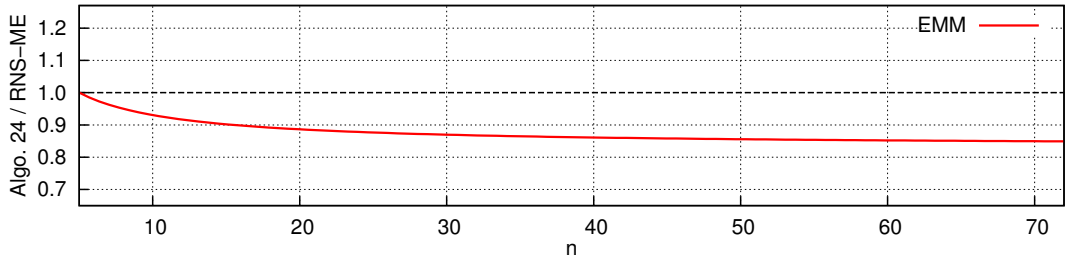

FIGURE 3.6 – Rapport du nombre d'EMM entre notre algorithme 24 et l'algorithme de l'état de l'art [48].

On rappelle tout d'abord que l'on a toujours $K_s < \left(\frac{3P}{M_a}\right)$ et $R_s < 2M_a$ (à cause de l'approximation dans l'extension de base de Kawamura *et al.* [64]). Ensuite, les bases $\mathcal{B}_a$, $\mathcal{B}_b$ et $\mathcal{B}_c$ sont choisies pour satisfaire les conditions d'utilisation de l'algorithme MR, c'est-à-dire telles que $M_c > 3P$ et $M_aM_b > 3P$. On obtient alors la borne suivante sur U_2 :

$$\begin{aligned} U_2 &= K_s\left(K_s|M_a^2G|_P + R_s|2M_aG|_P\right) + R_s^2|G|_P \\ &\leqslant K_s^2P + K_sR_sP + R_s^2P \\ &< \frac{9P^3}{M_a^2} + 6P^2 + M_a^2P \\ &< P^2\left(\frac{3M_aM_b}{M_a^2} + 6\right) + M_a^2P \\ &< 12P^2 + M_a^2P \quad . \end{aligned}$$

De plus, on rappelle que $\mathcal{B}_a$ et $\mathcal{B}_b$ sont toutes deux composées de $n/2$ moduli de w bits, donc $\log_2 \frac{M_b}{M_a} < 1$, autrement dit $\frac{M_b}{M_a} < 2$. Nous allons maintenant déduire les différentes conditions sur les trois bases $\mathcal{B}_a$, $\mathcal{B}_b$ et $\mathcal{B}_c$. Tout d'abord, pour que la représentation de U_2 soit exacte, il faut $M_aM_bM_c > U_2$. De plus, pour pouvoir effectuer une autre itération de la boucle de l'algorithme 24, il faut qu'en sortie de l'itération on ait $S < 3P$. Comme pour l'analyse du SPRR, on déduit la condition sur M_c à partir du calcul effectué dans la réduction finale MR. La contrainte est donc $\frac{U_2+QP}{M_c} < 3P$. De plus :

$$\frac{U_2 + QP}{3P} < \frac{U_2}{3P} + \frac{2M_c}{3} < 4P + \frac{M_a^2 + 2M_c}{3} \quad , \tag{3.9}$$

car $Q < 2M_c$ (Q est la sortie de l'extension de base de Kawamura *et al.* à partir de M_c). Il suffit donc de choisir $M_c \geqslant 4P + \frac{M_a^2+2M_c}{3}$, c'est-à-dire $M_c \geqslant 12P + M_a^2$.

L'utilisation de l'algorithme 24 impose de stocker plus de pré-calculs que l'exponentiation de l'état de l'art [48] tout simplement parce que les appels à MR dans l'algorithme sont les mêmes que ceux de l'état de l'art, et requièrent exactement le même nombre de pré-calculs, c'est-à-dire $2n^2 + 10n$ EMW. De plus, il faut compter les pré-calculs pour la fonction Split. Ceux-ci ont été détaillés dans la table 3.2 pour le cas où $n_a = n_b = n_c = n/2$. Dans notre cas $n_c = n$, ce qui nous mène à $n^2 + 4n$ EMW. Enfin, le calcul de U_2 dans chacune des bases requiert $6n$ EMW pour les 3 valeurs pré-calculées dans chacune des bases. Nous obtenons donc un total de $3n^2 + 20n$ EMW contre $2n^2 + 10n$ EMW.

3.3.2 Autres algorithmes d'exponentiation

Notre algorithme est basé sur le fait que le motif $S^2G \bmod P$ se comporte très bien lorsqu'on le décompose avant de calculer la réduction, grâce au fait qu'il y a beaucoup de

Algorithme 25: Exponentiation RNS régulière revisitée.

Entrée : $k = (k_{\ell-1}, \ldots, k_1, k_0)_2$
Pré-calculs : $\overrightarrow{\left(M_{c,i}^{-1}P^{-1}|GM_a^2|_P\right)_c}$, $\overrightarrow{\left(M_{c,i}^{-1}P^{-1}|2GM_a|_P\right)_c}$, $\overrightarrow{\left(M_{c,i}^{-1}P^{-1}G\right)_c}$ dans $\mathcal{B}_c$
Pré-calculs : $\overrightarrow{\left(|GM_a^2|_P\right)_{a|b}}$, $\overrightarrow{\left(|2GM_a|_P\right)_{a|b}}$, $\overrightarrow{G_{a|b}}$ dans $\mathcal{B}_{a|b}$
Pré-calculs : $\overrightarrow{\left(M_{c,i}^{-1}P^{-1}|M_a^2|_P\right)_c}$, $\overrightarrow{\left(M_{c,i}^{-1}P^{-1}|2M_a|_P\right)_c}$, $\overrightarrow{\left(M_{c,i}^{-1}P^{-1}\right)_c}$ dans $\mathcal{B}_c$
Pré-calculs : $\overrightarrow{\left(|M_a^2|_P\right)_{a|b}}$, $\overrightarrow{\left(|2M_a|_P\right)_{a|b}}$, $\overrightarrow{1_{a|b}}$ dans $\mathcal{B}_{a|b}$
Sortie : $\overrightarrow{|G^k|_P}$ dans $\mathcal{B}_{a|b|c}$

$\overrightarrow{S_{a|b|c}} \leftarrow \overrightarrow{1}$
pour i **de** $\ell - 1$ **à** 0 **faire**
  $(\overrightarrow{K_s}, \overrightarrow{R_s}) \leftarrow \texttt{Split}(\overrightarrow{S})$
  si $k_i = 0$ **alors**
    $\overrightarrow{S_{a|b}} \leftarrow \overrightarrow{(K_s)_{a|b}} \times \left(\overrightarrow{(K_s)_{a|b}} \times \overrightarrow{\left(|M_a^2|_P\right)_{a|b}} + \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{\left(|2M_a|_P\right)_{a|b}}\right)$
    $\overrightarrow{S_{a|b}} \leftarrow \overrightarrow{S_{a|b}} + \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{1_{a|b}}$
    $\overrightarrow{S_c} \leftarrow \overrightarrow{(K_s)_c} \times \overrightarrow{\left(M_{c,i}^{-1}P^{-1}|M_a^2|_P\right)_c} + \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{\left(M_{c,i}^{-1}P^{-1}|2M_a|_P\right)_c}$
    $\overrightarrow{S_c} \leftarrow \overrightarrow{(K_s)_c} \times \overrightarrow{S_c} + \overrightarrow{(R_s)_c} \times \overrightarrow{(R_s)_c} \times \overrightarrow{\left(M_{c,i}^{-1}P^{-1}\right)_c}$
  sinon
    $\overrightarrow{S_{a|b}} \leftarrow \overrightarrow{(K_s)_{a|b}} \times \left(\overrightarrow{(K_s)_{a|b}} \times \overrightarrow{\left(|GM_a^2|_P\right)_{a|b}} + \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{\left(|2GM_a|_P\right)_{a|b}}\right)$
    $\overrightarrow{S_{a|b}} \leftarrow \overrightarrow{S_{a|b}} + \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{G_{a|b}}$
    $\overrightarrow{S_c} \leftarrow \overrightarrow{(K_s)_c} \times \overrightarrow{\left(M_{c,i}^{-1}P^{-1}|GM_a^2|_P\right)_c} + \overrightarrow{(R_s)_{a|b}} \times \overrightarrow{\left(M_{c,i}^{-1}P^{-1}|2GM_a|_P\right)_c}$
    $\overrightarrow{S_c} \leftarrow \overrightarrow{(K_s)_c} \times \overrightarrow{S_c} + \overrightarrow{(R_s)_c} \times \overrightarrow{(R_s)_c} \times \overrightarrow{\left(M_{c,i}^{-1}P^{-1}G\right)_c}$
  $\overrightarrow{S_{a|b|c}} \leftarrow \texttt{MR}(\overrightarrow{S_c}, \overrightarrow{S_{a|b}})$
retourner $\overrightarrow{S_{a|b|c}}$

réutilisations (un carré et une multiplication par une constante). Pour d'autres algorithmes, comme l'algorithme 5 carré et multiplication poids faibles en tête, on ne multiplie pas par une constante ce qui, semble-t-il, rend inapplicable l'astuce que nous avons utilisé ici. Il en va de même pour l'échelle de Montgomery, qui peut profiter du `SPRR` pour l'accélérer mais pas du motif ici présent.

On peut par contre adapter notre algorithme pour être régulier, en effectuant une multiplication et un carré à chaque fois. L'algorithme 25 présente un exemple d'adaptation de notre précédent algorithme 24, en version régulière. On utilise en fait la même méthode que pour calculer $S^2G \bmod P$ pour calculer $S^2 \cdot 1 \bmod P$. On remarque alors qu'un tel algorithme coûterait $4n^2 + 8n$ avec la multiplication RNS de l'état de l'art (coût de 2 `MM`) par itération de boucle, contre $3n^2 + 13n$ pour l'algorithme 25 (une réduction de 22.5 % pour RSA-2048 avec $n = 66$). L'algorithme 25 requiert $6n$ `EMW` de plus que sa version non régulière, c'est-à-dire $3n^2 + 26n$ `EMW` en tout.

Enfin, il est aussi possible d'adapter les algorithmes fenêtrés qui utilisent $S^2G \bmod P$,

comme la version exponentiation de l'algorithme 7 présenté dans la section 1.1.4.

3.4 Conclusion

Dans ce chapitre a été proposée une nouvelle façon d'aborder la multiplication modulaire RNS, dont la caractéristique principale est de pouvoir accélérer les calculs lorsque les opérandes sont réutilisés plusieurs fois, grâce à des décompositions de ces opérandes. Les cas les plus simples de réutilisation sont les multiplications par une constante et les carrés.

Nous avons tout d'abord proposé un algorithme de multiplication, permettant l'utilisation d'un nombre réduit de moduli ($3n/2$ au lieu de $2n$ pour l'état de l'art). Ainsi, l'algorithme de multiplication proposé réduit le nombre de pré-calculs de 20 % à 25 % pour les paramètres étudiés. De plus, le nombre d'opérations élémentaires peut être réduit jusqu'à 10 % pour des applications cryptographiques sur de grands corps ou entiers (ECC et DH). Ainsi, nous nous attendons à ce que le coût d'une implantation matérielle de l'algorithme soit réduit par rapport à la solution de l'état de l'art sur des applications cryptographiques sur de grands paramètres. La contrainte de cet algorithme est qu'il nécessite une certaine condition sur le corps de base.

En reprenant certaines idées de l'algorithme de multiplication modulaire, il est possible de trouver certaines séquences de calcul très efficaces lorsqu'il y a des réutilisations d'opérandes. Nous avons alors proposé un algorithme d'exponentiation qui requiert moins de multiplications que l'algorithme de l'état de l'art. De plus, cet algorithme ne nécessite aucune condition sur le corps ou l'anneau sur lequel on calcule, il est donc utilisable pour les calculs RSA. Nous obtenons une réduction du nombre de multiplications élémentaires de 15 % pour RSA 2048 bits. Une version régulière de l'algorithme est aussi proposée, réduisant de 22 % le nombre de multiplications élémentaires, toujours pour RSA 2048 bits, par rapport à un algorithme de type *double-and-add always* ou échelle de Montgomery.

De futurs travaux porteront sur une étude plus approfondie des motifs qui peuvent être accélérés en RNS et sur une implantation matérielle complète, pour étudier plus précisément les gains et les surcoûts obtenus.

Chapitre 4

Multiplication modulaire RNS mono-base

Dans ce chapitre, un autre algorithme de multiplication modulaire RNS est proposé. Ce nouvel algorithme est applicable pour la multiplication modulo P, pour P bien choisi, dans un contexte de cryptographie sur courbes elliptiques. À notre connaissance, c'est le premier algorithme effectuant une multiplication modulaire sur seulement n moduli, c'est à dire sur une seule base dans les notations classiques. Le nombre d'opérations élémentaires est fortement réduit par rapport à la multiplication RNS de l'état de l'art et la multiplication `SPRR` du chapitre 3. On obtient de bons résultats même pour des petites valeurs de n. Ces travaux sont en cours, seuls des premiers résultats d'implantation matérielle sont fournis.

4.1 La multiplication modulaire RNS à base unique `SBMM`

Le nouvel algorithme que nous proposons utilise une variation de la représentation RNS, où un élément de $\mathbb{F}_P$, noté X, n'est plus représenté par $\overrightarrow{X}$, mais par $\overrightarrow{K_x}$ et $\overrightarrow{R_x}$ avec la relation $\overrightarrow{X} = \overrightarrow{K_x M_a + R_x}$. Nous retrouvons en quelque sorte la décomposition proposée dans le chapitre 3 pour la multiplication `SPRR`. Dans le chapitre 3, on utilisait cette décomposition pour factoriser certains calculs internes à la multiplication modulaire, les entrées et sorties étant en représentation RNS classique. Dans notre nouvelle proposition, nous représentons les éléments de $\mathbb{F}_P$ uniquement grâce à ces décompositions. Un élément de $\mathbb{F}_P$, noté X, sera donc maintenant représenté par la paire de vecteurs RNS $(\overrightarrow{K_x}, \overrightarrow{R_x})$. Chacun des vecteurs $\overrightarrow{K_x}$ et $\overrightarrow{R_x}$ est défini sur n moduli, au lieu d'avoir un seul vecteur RNS, $\overrightarrow{X}$, sur $2n$ moduli comme dans l'état de l'art.

Notre nouvel algorithme de multiplication modulaire, appelé `SBMM` pour *single base modular multiplication*, est présenté algorithme 26. Avant de détailler les conditions sur ses entrées, sorties et évaluer son coût, nous allons présenter les idées sous-jacentes. Tout d'abord, par soucis de clarté, nous allons découper la base RNS en deux demi-bases $\mathcal{B}_a$ et $\mathcal{B}_b$ de $n/2$ moduli (un peu comme pour le `SPRR` sauf qu'il n'y a maintenant plus que 2 demi-bases). Nous rappelons que M_a désigne le produit des éléments de $\mathcal{B}_a$ (ici produit de $n/2$ moduli). La première idée importante de l'algorithme est l'utilisation de cette représentation décomposée, nous allons voir ci-dessous comment l'algorithme tire parti de celle-ci. La deuxième idée de notre nouvel algorithme `SBMM` est de lier M_a et P via la condition $M_a^2 = P + c$, avec c très petit. Les meilleurs résultats sont obtenus avec $M_a^2 = P + 2$, nous allons nous tenir à ce cas pour la présentation de l'algorithme. Pour c quelconque,

Algorithme 26: Multiplication modulaire RNS mono-base `SBMM`.

Paramètres : $\mathcal{B}_a$ tel que $M_a^2 = P + 2$ et $\mathcal{B}_b$ tel que $M_b > 6M_a$

Entrées : $\overrightarrow{(K_x)_{a|b}}$, $\overrightarrow{(R_x)_{a|b}}$, $\overrightarrow{(K_y)_{a|b}}$, $\overrightarrow{(R_y)_{a|b}}$ avec K_x, R_x, K_y, $R_y < M_a$

Sorties : $\overrightarrow{(K_z)_{a|b}}$, $\overrightarrow{(R_z)_{a|b}}$ avec $K_z < 5M_a$ et $R_z < 6M_a$

1 $\overrightarrow{U_{a|b}} \leftarrow \overrightarrow{2K_xK_y + R_xR_y}$

2 $\overrightarrow{V_{a|b}} \leftarrow \overrightarrow{K_xR_y + R_xK_y}$ */*astuce de Karatsuba-Ofman utilisable ici*/*

3 $\left(\overrightarrow{(K_u)_{a|b}}, \overrightarrow{(R_u)_{a|b}}\right) \leftarrow$ `CSplit`$(\overrightarrow{U_{a|b}})$

4 $\left(\overrightarrow{(K_v)_{a|b}}, \overrightarrow{(R_v)_{a|b}}\right) \leftarrow$ `CSplit`$(\overrightarrow{V_{a|b}})$

5 **retourner** $\left(\overrightarrow{(K_u + R_v)_{a|b}}, \overrightarrow{(2 \cdot K_v + R_u)_{a|b}}\right)$

une démonstration générale sera proposée dans la section 4.2.1. Cette contrainte étant très forte, elle ne peut être appliquée que dans le cadre d'ECC, et pas sur le logarithme discret à la différence de `SPRR`. Nous allons voir que cette condition va permettre d'effectuer la multiplication modulaire RNS de manière analogue à ce qui est proposé dans les standards ECC du NIST [91] pour la représentation binaire classique, c'est-à-dire analogue à la multiplication modulo un nombre premier pseudo-Mersenne en base 2.

L'algorithme `SBMM` vient du constat suivant. Supposons que X et Y soient deux éléments de $\mathbb{F}_P$, et soient (K_x,R_x) et (K_y,R_y) leurs décompositions en quotient/reste par M_a. On rappelle que M_a est le produit des éléments de $\mathcal{B}_a$. Alors en calculant le produit XY, on obtient :

$$\begin{aligned} XY = &\quad K_xK_yM_a^2 + (K_xR_y + K_yR_x)M_a + R_xR_y \mod P && (4.1)\\ \equiv &\quad 2\,K_xK_y + R_xR_y + (K_xR_y + K_yR_x)M_a \mod P && (4.2)\\ \equiv &\quad 2\,K_xK_y + R_xR_y + (K_xK_y + R_xR_y - (K_x - R_x)(K_y - R_y))M_a \mod P && (4.3)\\ \equiv &\quad U + VM_a \mod P && (4.4) \end{aligned}$$

grâce au fait que $M_a^2 \bmod P = 2$.

Ce résultat ressemble à l'équation 3.2 obtenue au chapitre 3, mais avec des contraintes différentes. Le passage de l'équation 4.2 à l'équation 4.3 est juste une application de l'astuce de Karatsuba-Ofman [63]. On remarque qu'on ne peut pas calculer l'équation 4.4 avec seulement n moduli car VM_a est un entier de $\frac{3\ell}{2}$ bits (et nécessite donc $\frac{3n}{2}$ moduli pour être représenté). Par contre, U et V sont eux représentables dans notre base $\mathcal{B}_{a|b}$, on peut donc les calculer avec $U = (2K_xK_y + R_xR_y)$ et $V = (K_xR_y + K_yR_x)$. Cela vient du fait que K_x, K_y, R_x, $R_y < M_a$ et $M_b > 6M_a$, on obtient donc U, $V < 3M_a^2 < M_aM_b$. Le calcul de U et V ne requiert que des opérations efficaces en RNS (multiplications et additions). Ensuite, on peut découper U et V grâce à l'algorithme 27, noté `CSplit` pour *compact split*, qui exécute en fait la même fonction que `Split` du chapitre 3, mais pour seulement 2

demi-bases. En utilisant une nouvelle fois la propriété $M_a^2 = P + 2$ on obtient finalement :

$$\begin{aligned} XY &\equiv U + VM_a \mod P \\ &\equiv K_u M_a + R_u + K_v M_a^2 + R_v M_a \mod P \\ &\equiv (K_u + R_v) M_a + R_u + 2K_v \mod P \\ &\equiv K_z M_a + R_z \mod P. \end{aligned}$$

Dans cette équation, $K_z < 4M_a$ et $R_z < 5M_a$, ce qui implique que $K_z M_a + R_z < 5\ P$. Nous avons ainsi réduit le produit, ou presque, car quelques soustractions par P peuvent être requises. De même, K_z et R_z peuvent être un peu plus grands que les entrées K_x, R_x, K_y et R_y, à quelques soustractions par M_a près. La démonstration de ces bornes et les conséquences de la différence entre les tailles des entrées et celles des sorties seront détaillées dans la section 4.2.3.

Algorithme 27: Étape de décomposition compacte `CSplit`.

Entrée : $\overrightarrow{X_{a|b}}$
Pré-calculs : $\overrightarrow{\left(M_a^{-1}\right)_b}$
Sorties : $\overrightarrow{(K_x)_{a|b}}$, $\overrightarrow{(R_x)_{a|b}}$ avec $\overrightarrow{X_{a|b}} = \overrightarrow{(K_x)_{a|b}} \times \overrightarrow{(M_a)_{a|b}} + \overrightarrow{(R_x)_{a|b}}$

1 $\overrightarrow{(R_x)_b} \leftarrow \texttt{BE}\left(\overrightarrow{(R_x)_a}, \mathcal{B}_a, \mathcal{B}_b\right)$
2 $\overrightarrow{(K_x)_b} \leftarrow \left(\overrightarrow{X_b} - \overrightarrow{(R_x)_b}\right) \times \overrightarrow{\left(M_a^{-1}\right)_b}$
3 **si** $\overrightarrow{(K_x)_b} = \overrightarrow{-1}$ **alors**
4 $\quad \overrightarrow{(K_x)_b} \leftarrow \overrightarrow{0}$ /* *correction d'une erreur avec la BE de Kawamura* */
5 $\quad \overrightarrow{(R_x)_b} \leftarrow \overrightarrow{(R_x)_b} - \overrightarrow{(M_a)_b}$
6 $\overrightarrow{(K_x)_a} \leftarrow \texttt{BE}\left(\overrightarrow{(K_x)_b}, \mathcal{B}_b, \mathcal{B}_a\right)$
7 **retourner** $\overrightarrow{(K_x)_{a|b}}$, $\overrightarrow{(R_x)_{a|b}}$

Pour résumer, on peut considérer que les valeurs sont représentées en numération de position de base M_a, avec une « partie haute » K_x et une « partie basse » R_x, K_x et R_x étant représentées en RNS en base $\mathcal{B}_{a|b}$ par $\overrightarrow{(K_x)}_{a|b}$ et $\overrightarrow{(R_x)}_{a|b}$. On applique ensuite deux fois la propriété $P+2 = M_a^2$, qu'on peut qualifier de propriété « à la Mersenne », pour réduire notre valeur. La similitude avec la réduction modulo un premier pseudo-Mersenne vient du fait que, dans les 2 cas, la multiplication modulaire se résume à découper le résultat du produit en utilisant la forme de P, puis à additionner les morceaux obtenus. Dans notre algorithme `SBMM`, deux découpages intermédiaires, pour U et V, en partie haute et partie basse sont nécessaires entre les deux applications de $P + 2 = M_a^2$. Une grosse différence avec une multiplication modulo un premier pseudo-Mersenne en représentation classique est, qu'ici, ce n'est pas la partie multiplication qui est coûteuse, mais le découpage en partie haute et partie basse (alors qu'en représentation binaire classique, cette opération est gratuite). Dans l'algorithme 26, les lignes 1 et 2 calculent U et V : on applique une première fois $M_a^2 = P + 2$. Ensuite les lignes 3 et 4, découpent U et V grâce à la fonction `CSplit`, ce qui représente 4 extensions de base. Cependant, ces 4 extensions de bases ne portent que sur les deux demi-bases $\mathcal{B}_a$ et $\mathcal{B}_b$. Puisque le coût d'une extension est quadratique en n, le coût

de ces 4 petites extensions est équivalent au coût d'une seule grande entre deux bases de n moduli chacune. Nous allons voir dans la section 4.2.4 que notre algorithme coûte bien moins de multiplications élémentaires `EMM` que l'algorithme de l'état de l'art et qu'il en va de même pour le nombre de pré-calculs `EMW`.

4.2 Analyse de l'algorithme `SBMM`

4.2.1 Généralisation du paramètre c

Dans l'analyse de cet algorithme 26, nous allons étudier le cas plus général $P+c = M_a^2$, avec c un petit nombre positif, et démontrer les bornes que cela induit. Tout d'abord, on remarque que c ne doit pas être un carré, sinon

$$P = M_a^2 - c = (M_a + \sqrt{c})(M_a - \sqrt{c}) \, ,$$

c'est à dire P n'est pas premier. Le plus petit c possible est 2, avec M_a impair (c'est à dire $\mathcal{B}_a$ n'est composée que d'éléments impairs). Nous n'étudierons pas ici le cas c négatif, car il introduit un certain nombre de valeurs négatives qu'il faut pouvoir gérer correctement en RNS. Le cas c négatif semble donc moins intéressant. Pour commencer, nous allons dans un premier temps supposer que les extensions de base sont exactes, par exemple en utilisant les extensions de base via MRS. Les valeurs qui sont transférées d'une base à l'autre, comme R_u ou K_u, sont donc transférées sans erreur d'approximation.

En suivant le même cheminement que dans la section précédente, mais pour un c quelconque, on obtient :

$$\begin{aligned} XY &\equiv c\,K_xK_y + R_xR_y + (K_xR_y + K_yR_x)M_a \mod P \\ &\equiv U + V M_a \mod P \\ &\equiv (K_u + R_v)M_a + R_u + cK_v \mod P \\ &\equiv K_zM_a + R_z \mod P \quad . \end{aligned}$$

Ici, seules les expressions de U et R_z changent, devenant $U = c\,K_xK_y + R_xR_y$ et $R_z = R_u + cK_v$. Nous allons maintenant démontrer les bornes sur M_a et M_b. Pour représenter U et V sur la base $\mathcal{B}_{a|b}$, il faut que U, $V < M_aM_b$. On peut borner U et V par :

$$\begin{aligned} U = &\quad c\,K_xK_y + R_xR_y < cM_a^2 + M_a^2 \leqslant (c+1)M_a^2 \\ V = &\quad K_xR_y + K_yR_x < 2M_a^2 \quad . \end{aligned}$$

Il suffit donc que $M_b > (c+1)M_a$ pour représenter U et V (on rappelle que $c \geqslant 2$). Il nous faut maintenant être en mesure de représenter les valeurs en sortie, K_z et R_z. On rappelle que K_u est le quotient de U par M_a, et que $U < (c+1)M_a^2$, on a donc $K_u < (c+1)M_a$. Par définition, R_u étant le reste de la division on a $R_u < M_a$. De même on a $K_v < 2M_a$ et $R_v < M_a$. On en conclut :

$$\begin{aligned} K_z = &\quad K_u + R_v < (c+2)M_a \\ R_z = &\quad R_u + cK_v < (2c+1)M_a \quad . \end{aligned}$$

On va donc choisir $\mathcal{B}_b$ pour avoir $M_b > (2c+1)M_a$. En prenant $c = 2$, on obtient $K_z < 4M_a$ et $R_z < 5M_a$. On ne retrouve pas exactement les bornes du cas $c = 2$ de l'algorithme 26 pour les sorties K_z et R_z car celles-ci sont calculées pour l'extension de base de Kawamura *et al.* [64]. On remarque par contre que les éléments en sortie sont plus grands que ceux en entrée : pour limiter cette croissance on choisira c le plus petit possible. Avec une demi-base $\mathcal{B}_a$ ne contenant que des éléments impairs, on peut facilement trouver des bases avec $M_a^2 = P + 2$ et P premier. Si on préfère une base $\mathcal{B}_a$ avec un moduli pair, on ne peut alors pas avoir mieux que $M_a^2 = P + 3$. Avoir un peu de latitude sur c peut être intéressant pour avoir un plus grand choix de bases. On peut même encore augmenter ces choix en cherchant des bases avec $M_a^2 = \mu P + c$ avec μ très petit. Bien sûr, en se donnant plus de choix pour P on est contraint d'agrandir encore la base M_b.

4.2.2 Utilisation de l'extension de base de Kawamura *et al.*

Considérons maintenant l'utilisation de l'extension de base de Kawamura *et al.* [64]. Comme pour le SPRR et pour MM, on ne peut pas effectuer de manière exacte la première extension de base en utilisant l'extension de base de Kawamura *et al.* [64], nous sommes dans le cas du théorème 5. La stratégie sera ici la même que pour le SPRR. Si il y a une erreur, elle se produira lors du calcul de $\overrightarrow{(R_u)_b}$ (respectivement $\overrightarrow{(R_v)_b}$) à la ligne 1 de l'algorithme 27. Supposons que l'extension de base produise par erreur $\overrightarrow{(R'_u)_b}$ avec $\overrightarrow{(R'_u)_b} = \overrightarrow{(R_u + M_a)_b}$ (resp. $\overrightarrow{(R'_v)_b} = \overrightarrow{(R_v + M_a)_b}$) dans la base $\mathcal{B}_b$. Alors, le calcul à la ligne 2 de l'algorithme 27 donne :

$$\overrightarrow{(K'_u)_b} = \left(\overrightarrow{U_b} - \overrightarrow{(R'_u)_b}\right) \times \overrightarrow{\left(M_a^{-1}\right)_b} = \left(\overrightarrow{U_b} - \overrightarrow{(R_u)_b} - \overrightarrow{(M_a)_b}\right) \times \overrightarrow{\left(M_a^{-1}\right)_b} = \overrightarrow{(K_u - 1)_b} \quad .$$

Le calcul ne change donc rien au fait que $U = K'_u M_a + R'_u$ dans la base $\mathcal{B}_b$. Par contre, si $K_u = 0$, il est possible que l'on ait $K'_u = -1$, impliquant une réduction implicite par M_b dans la demi-base $\mathcal{B}_b$, ce qui empêche de retourner le bon résultat lors de la deuxième extension de base à la ligne 3 de l'algorithme 27. Pour contrer ce problème, il suffit de vérifier que $K'_u \neq -1$ dans la base $\mathcal{B}_b$. Si $K'_u = -1$ dans $\mathcal{B}_b$, alors on corrige en effectuant $\overrightarrow{(K_u)_b} = 0$ et $\overrightarrow{(R_u)_b} = \overrightarrow{(R'_u + M_a)_b}$ dans la seconde base après la ligne 2 de l'algorithme 27 et avant l'extension de base. Le comparateur utilisé pour $K'_u \neq -1$ est déjà présent en matériel si on utilise l'algorithme d'inversion PM-MI (cf. chapitre 2). Nous n'avons pas approfondi d'autres solutions pour régler ce cas spécial car il n'arrivera pas, en pratique, dans la plupart des applications. En effet, supposons que X soit un élément quelconque de $\mathbb{F}_P$ d'au moins 160 bits, ce qui est le plus petit standard du NIST [91]. Alors, avoir $K_x = 0$ est équivalent à avoir $X < M_a$, avec $\log_2 M_a \approx 80$ bits au minimum, plus généralement $\log_2 M_a \approx \ell/2$. On a donc une probabilité $\frac{1}{2^{80}}$ de tomber sur $K_x = 0$. En dehors de la gestion de ce cas très particulier, il est aussi à noter que les valeurs K_z et R_z sont potentiellement plus grandes en utilisant une extension de base approchée. Plus exactement, avec l'extension de base de Kawamura *et al.*, on obtient les bornes R_u, $R_v < 2M_a$, ce qui implique :

$$\begin{aligned} K_z &= K_u + R_v < (c+3)M_a \\ R_z &= R_u + cK_v < (2c+2)M_a \quad . \end{aligned}$$

On retrouve bien les bornes présentées dans l'algorithme 26 sur K_z et R_z.

4.2.3 Compression des sorties de l'algorithme

Nous allons maintenant discuter du problème de la taille des sorties de l'algorithme. En effet, par exemple si $c = 2$, les entrées de l'algorithme sont inférieures à M_a alors qu'en sortie on a $K_z < 5M_a$ et $R_z < 6M_a$. Ainsi en réappliquant successivement `SBMM`, on obtient $K_{z,2} < 87M_a$ et $R_{z,2} < 121M_a$ puis $K_{z,3} < 29780M_a$ et $R_{z,2} < 42109M_a$ etc. La taille des opérandes va ainsi croître de façon exponentielle si on applique récursivement et sans contrôle l'algorithme `SBMM`.

Une première façon de procéder pour ce contrôle est d'effectuer une nouvelle réduction après un certain nombre d'itérations. Nous verrons dans l'exemple proposé à la fin de cette sous-section que dans les calculs nécessaires aux doublements et additions de points d'ECC, un certain nombre de multiplications sont complètement indépendantes. Ainsi, si par exemple 4 multiplications indépendantes sont effectuées, leurs sorties ont toutes la même taille. On doit donc déduire du flot de calcul la taille maximale des sorties de l'algorithme `SBMM` afin de choisir M_b suffisamment grand pour garantir $X = K_x M_a + R_x < M_a M_b$. Comme la taille des sorties croît très vite, seulement 2 ou 3 multiplications successives peuvent être effectuées avant une réduction modulaire de contrôle. Pour effectuer cette réduction à partir de notre algorithme, il suffit de l'appliquer sur la valeur à réduire, par exemple (K_z,R_z), et la valeur 1, représentée par (0,1).

Algorithme 28: Compression d'une valeur représentée par (K, R).

Entrées : $\overrightarrow{K_{a|b|m_\gamma}}$ et $\overrightarrow{R_{a|b|m_\gamma}}$ avec K, $R < m_\gamma - 1$
Pré-calcul : $\left|M_a^{-1}\right|_{m_\gamma}$
Sorties : $\overrightarrow{(K_c)_{a|b|m_\gamma}}$, $\overrightarrow{(R_c)_{a|b|m_\gamma}}$ avec K_c, $R_c < 2M_a + 6$

1 $|R_k|_{m_\gamma} \leftarrow \texttt{BE}\left(\overrightarrow{K_a}, \mathcal{B}_a, m_\gamma\right)$ /* $\overrightarrow{(R_k)_a} = \overrightarrow{K_a}$ */
2 $K_k \leftarrow \left|(K - R_k)M_a^{-1}\right|_{m_\gamma}$
3 $\overrightarrow{(R_k)_b} \leftarrow \overrightarrow{K_b} - \overrightarrow{(K_k)_b} \times \overrightarrow{(M_a)_b}$
4 $|R_r|_{m_\gamma} \leftarrow \texttt{BE}\left(\overrightarrow{R_a}, \mathcal{B}_a, m_\gamma\right)$ /* $\overrightarrow{(R_r)_a} = \overrightarrow{R_a}$ */
5 $K_r \leftarrow \left|(R - R_r)M_a^{-1}\right|_{m_\gamma}$
6 $\overrightarrow{(R_r)_b} \leftarrow \overrightarrow{R_b} - \overrightarrow{(K_r)_b} \times \overrightarrow{(M_a)_b}$
7 **return** $\overrightarrow{(K_r + R_k)_{a|b|m_\gamma}}$, $\overrightarrow{(R_r + 2K_k)_{a|b|m_\gamma}}$

Une autre méthode, plus rapide mais nécessitant un peu de matériel supplémentaire, est présentée dans l'algorithme 28. Il suffit de remarquer que les valeurs en sortie de l'algorithme K_z et R_z, sont proches de M_a. En effet, on a $K_z < 5M_a$ et $R_z < 6M_a$. Soit $(\overrightarrow{K}_{a|b}, \overrightarrow{R}_{a|b})$ une valeur en sortie de l'algorithme 26. Supposons qu'on décompose K en $K = K_k M_a + R_k$ et R en $R = K_r M_a + R_r$. On a alors $K_k < 5$ et $K_r < 6$. On peut donc calculer une extension de base à partir de $\mathcal{B}_a$ vers un modulo supplémentaire m_γ, premier avec M_a, puis obtenir la valeur exacte de ces quotients, du moment que $m_\gamma > 6$. Il faut quand même avoir $K \bmod m_\gamma$ et $R \bmod m_\gamma$ dans ce modulo supplémentaire avant de pouvoir faire la division. On peut choisir m_γ comme une petite puissance de 2, et choisir un élément de $\mathcal{B}_b$, disons $m_{b,1}$, égal à 2^w. On peut ensuite déduire directement la valeur de $K \bmod m_\gamma$ et $R \bmod m_\gamma$ à partir des bits de poids faibles modulo $m_{b,1}$. On remarque que puisque K_r et K_k sont très petits, nous n'avons pas besoin d'extension de base pour les représenter dans

Opération	$\mathbf{P_1} + \mathbf{P_2}$	$2\ \mathbf{P_1}$
Formules	$A = Z_1X_2 + Z_2X_1$	$E = Z_1^2$
	$B = 2X_1X_2$	$F = 2X_1Z_1$
	$C = 2Z_1Z_2$	$G = X_1^2$
	$D = aA + bC$	$H = -4bE$
	$Z_3 = A^2 - BC$	$I = aE$
	$X_3 = BA + CD + 2X_GZ_3$	$X_3 = FH + (G - I)^2$
		$Z_3 = 2F(G + I) - EH$

TABLE 4.1 – Formules optimisées RNS issues de [8] pour les courbes sous forme de Weierstrass courte, en coordonnées (X, Z) et adaptées à l'échelle de Montgomery.

$\mathcal{B}_a$ et $\mathcal{B}_b$. En effet, on a $(k_r)_{a,i} = (k_r)_{b,i} = K_r$ et $(k_k)_{a,i} = (k_k)_{b,i} = K_k$ pour tout i. Après avoir obtenu K_k et K_r, on peut utiliser le fait que $M_a^2 = 2 \bmod P$ pour obtenir :

$$\begin{aligned} X &\equiv K M_a + R \bmod P \\ &\equiv K_k M_a^2 + R_k M_a + K_r M_a + R_r \bmod P \\ &\equiv (R_k + K_r) M_a + 2K_k + R_r \bmod P \\ &\equiv K_c M_a + R_c \bmod P\,. \end{aligned}$$

La valeur retournée par la compression est donc (K_c, R_c). On note que R_k et R_r sont tous deux inférieurs à $2M_a$ si on utilise l'extension de base de Kawamura *et al.*, impliquant $K_c < 2M_a + 6$ et $R_c < 2M_a + 10$. Si deux valeurs (K_x, R_x) et (K_y, R_y) provenant d'une compression à l'aide de l'algorithme 28 sont utilisées en entrée de l'algorithme 26 de multiplication SBMM, on obtient alors en sortie $K_z < 14M_a$ et $R_z < 19M_a$ au lieu de $K_z < 5M_a$ et $R_z < 6M_a$. Pour pouvoir réutiliser l'algorithme de compression, il suffit de prendre $m_\gamma > 19$. Le plus simple est de prendre $m_\gamma = 32$, en choisissant la base $\mathcal{B}_a$ sans modulo pair. Pour ne pas avoir besoin d'utiliser systématiquement la fonction de compression, on peut augmenter la valeur de m_γ afin d'augmenter la correction maximale que l'on peut effectuer. Par exemple, si on enchaîne deux multiplications modulaires sans utiliser de compression, on obtient en sortie $K_z < 753M_a$ et $R_z < 1065M_a$. En prenant $m_\gamma = 2^{11} = 2048$, on peut se permettre de n'effectuer qu'une compression toutes les 2 multiplications. Nous allons voir maintenant un exemple pour lequel il est utile de n'effectuer qu'une compression toutes les 2 multiplications.

Un exemple d'utilisation de la fonction de compression est illustré à la figure 4.1. Dans cette exemple, on suppose l'exécution d'une addition de points suivie d'un doublement, avec les formules de la table 4.1 (issues de [8]). On utilise l'algorithme de multiplication `SBMM` pour calculer les différentes valeurs intermédiaires des formules, et le résultat est compressé en parallèle par la fonction de compression. Pour presque tous les calculs, l'algorithme `SBMM` reçoit en entrée des valeurs compressées. Il y a deux cas pour lesquels `SBMM` va recevoir en entrée des valeurs qui ne l'ont pas été. Le premier cas est le calcul de X_3 dans l'opération `ADD`. En effet, le calcul de X_3 dans ce cas requiert la valeur Z_3, qui n'a pas encore eu le temps d'être compressée : on est donc obligé de prendre directement le résultat de la multiplication `SBMM`. On se retrouve donc dans le cas où m_γ est choisi pour pouvoir enchaîner 2 multiplications `SBMM`, celle de Z_3 et celle de X_3. On compressera directement X_3 ensuite. On retrouve un cas similaire pour le calcul de X_3 dans le calcul `DBL`, où la

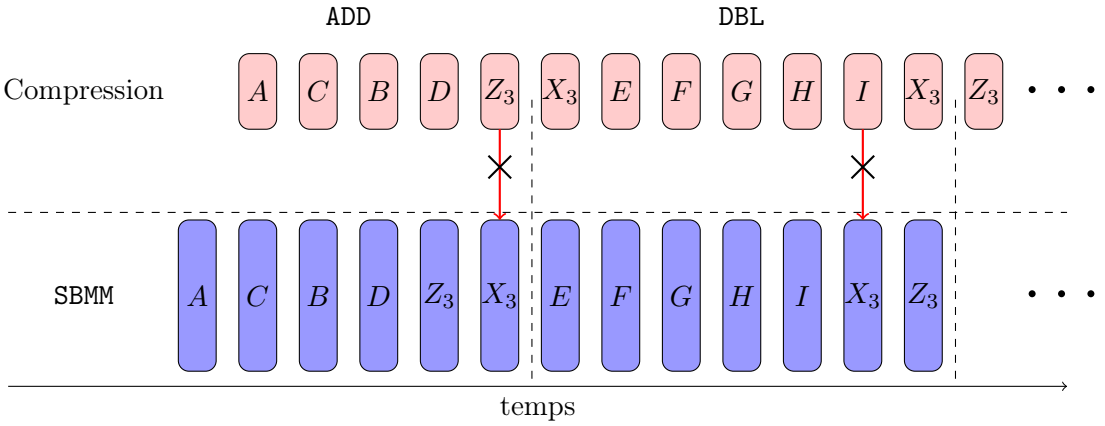

FIGURE 4.1 – Exemple fictif de flot d'exécution utilisant SBMM et la fonction de compression en parallèle, sur les formules d'addition et doublement de points définies à la table 4.1, avec 2 dépendances de données non satisfaites à temps entre la compression et SBMM.

valeur I, qui n'a pas encore été compressée, est utilisée. Finalement, cet exemple présente une façon d'effectuer le SBMM et la fonction de compression en parallèle, qui pourra être implantée dans le futur pour une multiplication scalaire complète.

4.2.4 Analyse des coûts en EMM et EMW

Pour commencer nous allons analyser le coût de l'algorithme 26 en nombre d'opérations. Nous allons nous contenter de compter ici les multiplications élémentaires EMM, comme sont faits habituellement les décomptes de l'état de l'art et comme ce qui a été fait pour le chapitre 3. Dans tous les cas, que ce soit pour notre algorithme ou pour l'état de l'art, les nombres d'EMM et d'EMA sont égaux (ou presque), car les deux algorithmes n'effectuent quasiment que des multiplications suivies d'accumulations. De plus, dans le cas de l'algorithme 26, nous n'allons pas compter les multiplications par 2 que l'on trouve aux lignes 1 et 5 car on peut les voir comme une simple addition.

Les lignes 1, 2 et 5 de l'algorithme SBMM coûtent $4n$ EMM, qui peuvent être réduites à $3n$ EMM en utilisant l'astuce de Karatsuba-Ofman [63] pour calculer les lignes 1 et 2. Le choix de l'utilisation de cette astuce dépend de l'architecture implantée, car elle introduit notamment des dépendances de données entre les lignes 1 et 2, alors qu'elles sont indépendantes telles qu'écrites dans l'algorithme SBMM. En plus de ces opérations, on effectue deux appels à CSplit. Le coût de CSplit est en fait le même que celui de la fonction Split, en considérant que les 2 premières bases ont le même nombre d'éléments égal à $n/2$, et que la troisième base n'a aucun élément. On obtient, pour chacun des appels de CSplit, un nombre d'EMM de :

$$n_a + n_a(n_b + n_c) + n_b + n_b n_a + n_c = 2\left(\frac{n^2}{4} + \frac{n}{2}\right) = \frac{n^2}{2} + n\,.$$

Le coût total de l'algorithme est donc $n^2 + 5n$ EMM pour effectuer une multiplication modulo P, contre $2n^2 + 4n$ avec l'algorithme de l'état de l'art [48]. Par contre, ce coût ne prend pas en compte une éventuelle compression des données en sortie. La fonction de compression proposée dans l'algorithme 28 effectue 2 extensions de base de $\mathcal{B}_a$ vers m_γ

aux lignes 1 et 4, puis 2 multiplications sur m_γ aux lignes 2 et 5, et finalement 2 multiplications sur $\mathcal{B}_b$. Les extensions de base coûtent $n/2$ EMM sur $\mathcal{B}_a$ et $n/2$ multiplications sur m_γ chacune. On compte ici les multiplications modulo m_γ à part, car généralement ces multiplications seront sur 5, 6 ou 12 bits alors que les EMM correspondront à des multiplications sur w bits (16 ou 32 par exemple). De plus, une EMM est une multiplication modulo un nombre pseudo-Mersenne, alors que m_γ est une puissance de 2 : la réduction modulaire est donc immédiate pour m_γ. On notera γEMM ces multiplications modulo m_γ. On obtient au total $2n$ EMM et $(n+2)$ γEMM. Le coût total de l'algorithme SBMM suivi de la fonction de compression est donc de n^2+7n EMM et $(n+2)$ γEMM contre $2n^2+4n$ EMM dans l'état de l'art. On a donc divisé par 2, environ, le nombre d'EMM par rapport à l'état de l'art.

Nous allons maintenant analyser la parallélisation que propose l'algorithme SBMM et la fonction de compression, car c'est une caractéristique essentielle de la représentation RNS. Dans l'état de l'art, on utilise généralement n unités arithmétiques, par exemple n Rowers, qui correspondent à un élément pour chacune des bases. Dans l'algorithme de l'état de l'art, on utilise uniquement des opérations sur des bases complètes de n moduli, qui se parallélisent donc très bien sur les n Rowers. Notre algorithme 26 calcule, lui, sur les 2 « demi-bases » $\mathcal{B}_a$ et $\mathcal{B}_b$ de $n/2$ moduli. Plus précisément, on trouve 2 cas distincts. Soit l'algorithme 26 effectue la même opération sur les deux bases en même temps, comme aux lignes 1, 2 et 5. Dans ce cas, on peut voir $\mathcal{B}_{a|b}$ comme une base complète de n moduli, et paralléliser sur les n Rowers comme d'habitude. Soit, les 2 appels à CSplit, lignes 3 et 4, n'effectuent leurs calculs que sur l'une des 2 demi-bases à la fois. Chacun des CSplit va donc opérer sur seulement $n/2$ moduli. Ce n'est en réalité pas un problème car les 2 appels à la fonction CSplit sont complètement indépendants : on peut donc les effectuer en parallèle, chacun sur $n/2$ moduli. Toutes les opérations de l'algorithme 26 sont donc parallélisables sur n Rowers facilement, comme pour l'algorithme de l'état de l'art MM [99] et ses améliorations [6, 48, 64].

La fonction de compression, présentée algorithme 28, est proposée pour une architecture où les calculs sur m_γ peuvent être effectués en parallèle des calculs sur les bases $\mathcal{B}_a$ et $\mathcal{B}_b$. Par exemple, en ayant une sorte de petit Rower supplémentaire, calculant modulo 2^6 ou 2^{12}. Ainsi, les lignes 2, 5, et les opérations sur m_γ à l'intérieur des extensions de base sont faites en parallèle des autres calculs sur les 2 bases $\mathcal{B}_a$ et $\mathcal{B}_b$. Ensuite, les calculs aux lignes 3 et 6, effectués sur la base $\mathcal{B}_b$, sont complètement indépendants : ils peuvent être effectués en même temps, chacun sur $n/2$ Rowers. Il en va de même pour les calculs sur $\mathcal{B}_a$ effectués par les deux BE aux lignes 1 et 4.

La table 4.2 présente le décompte des pré-calculs pour notre nouvel algorithme. Les pré-calculs du SBMM viennent tous des pré-calculs nécessaires à la fonction CSplit, qui utilise exactement les mêmes astuces que Gandino *et al.* [48] pour effectuer les extensions de base. Par exemple, les valeurs dans $\mathcal{B}_b$ sont multipliées par le pré-calcul $\overrightarrow{\left(T_b^{-1}\right)_b}$, de la même façon que Gandino *et al.* ont proposé dans [48]. La différence avec la contribution [48], c'est qu'ici on opère sur des demi-bases, et que nous ne faisons plus les calculs pour la réduction de Montgomery mais pour la fonction de décomposition CSplit. Les $n/2$ valeurs $\overrightarrow{\left(T_a^{-1}\right)_a}$ et les $n^2/4$ $\overrightarrow{\left(\frac{-1}{m_{a,i}M_{b,j}}\right)_b}$ permettent d'effectuer la première extension de base de CSplit, effectuée à la ligne 1 de l'algorithme CSplit, et permettent d'effectuer en même temps $-\overrightarrow{(R_x)_b} \times \overrightarrow{\left(M_a^{-1}\right)_b}$ de la ligne 2. Cette contraction d'opérations requiert le stockage de la valeur $\overrightarrow{(-M_a)_b}$ pour obtenir $\overrightarrow{(R_x)_b}$, car on obtient $-\overrightarrow{(R_x)_b} \times \overrightarrow{\left(M_a^{-1}\right)_b}$ à la sortie

SBMM	Compression
$\overrightarrow{\left(T_a^{-1}\right)_a}$: $n/2$	$\overrightarrow{\left(-M_a \times T_b^{-1}\right)_b}$: $n/2$
$\overrightarrow{\left(\frac{-1}{m_{a,i}M_{b,j}}\right)_b}$: $n^2/4$	$\left\|M_{a,i}\right\|_{m_\gamma}$: $n/2$*
$\overrightarrow{\left(\frac{M_{b,j}}{M_a}\right)_b}$: $n/2$	$\left\|-M_a\right\|_{m_\gamma}$: 1*
$\overrightarrow{\left(T_{b,i}\right)_a}$: $n^2/4$	$\left\|M_a^{-1}\right\|_{m_\gamma}$: 1*
$\overrightarrow{\left(-M_a\right)_b}$: $n/2$	
$\overrightarrow{\left(T_b^{-1}\right)_b}$: $n/2$	
$\overrightarrow{\left(-M_b\right)_a}$: $n/2$	
$\overrightarrow{\left(T_b\right)_b}$: $n/2$	
Total : $n^2/2 + 4n + 2$ EMW	

TABLE 4.2 – Décompte du nombre de pré-calculs à stocker en mots de w bits (EMW) de notre algorithme SPRR. Note * : les valeurs notées * sont en réalité plus petites que w bits.

Algorithme	coût en multiplications	coût en mémoire pré-calculée
MM	$2n^2 + 4n$ EMM	$2n^2 + 10n$ EMW
SBMM	$n^2 + 5n$ EMM ($\div 2$)	$\frac{n^2}{2} + 3n$ EMW ($\div 4$)
SBMM + compression	$(n^2 + 7n)$ EMM $+ (n+2)$ γEMM	$\frac{n^2}{2} + 4n + 2$ EMW

TABLE 4.3 – Comparaison du nombre de multiplications élémentaires et du nombre de mots mémoire élémentaires à stocker pour les algorithmes MM et SBMM, avec et sans compression des sorties.

de la première extension de base. Le vecteur $\overrightarrow{\left(\frac{M_{b,j}}{M_a}\right)_b}$ permet d'effectuer $\overrightarrow{X_b} \times \overrightarrow{\left(M_a^{-1}\right)_b}$. Les valeurs $\overrightarrow{\left(T_{b,i}\right)_a}$ et $\overrightarrow{\left(-M_b\right)_a}$ sont utilisées pour la deuxième extension de base de la fonction CSplit. Pour la compression, $\overrightarrow{\left(-M_a \times T_b^{-1}\right)_b}$ sert à calculer $\overrightarrow{(R_k)_b}$ et $\overrightarrow{(R_r)_b}$ aux lignes 3 et 6 de l'algorithme 28. Si on stocke $\overrightarrow{\left(-M_a \times T_b^{-1}\right)_b}$ plutôt que $\overrightarrow{\left(-M_a\right)_b}$, c'est parce que dans la seconde demi-base $\mathcal{B}_b$, les valeurs sont toutes multipliées par $\overrightarrow{\left(T_b^{-1}\right)_b}$ car on utilise la représentation de Gandino *et al.* [48]. Ensuite, les $n/2$ valeurs $\left|M_{a,i}\right|_{m_\gamma}$ sont nécessaires pour calculer le CRT modulo m_γ, et $\left|-M_a\right|_{m_\gamma}$ permet de corriger le calcul du CRT avec l'astuce de Kawamura *et al.* [64]. Enfin, $\left|M_a^{-1}\right|_{m_\gamma}$ permet de calculer K_k et K_r dans l'algorithme 28. La table 4.3 résume et compare les coûts en calcul et en mémoire entre l'algorithme de l'état de l'art MM et l'algorithme SBMM, avec et sans compression des sorties de celui-ci.

4.3 Implantation FPGA

Nous allons présenter dans cette section nos premiers résultats d'implantation. Ce sont des résultats partiels, car les implantations ont été effectuées durant la rédaction du document de thèse. Ils ne sont pas encore suffisamment optimisés, l'architecture devrait notamment être modifiée pour mieux profiter de notre algorithme. Nous nous sommes ici comparés aux multiplieurs modulaire RNS, tels qu'implantés pour le chapitre 2 sur l'inversion modulaire RNS `PM-MI`. Des premiers gains intéressants sont observés et présentés ci-dessous.

4.3.1 Architecture implantée

Tout d'abord, afin d'effectuer une implantation matérielle, il faut générer les différents paramètres de notre représentation. Pour des paramètres w et n choisis, nous avons tiré au hasard $n/2$ éléments parmi les nombres pseudo-Mersenne impairs de la forme $m_{a,i} = 2^w - h_{a,i}$, avec $h_{a,i} < 2^{\frac{w}{2}}$. Avant d'inclure un nouvel élément dans la base, on vérifie qu'il soit bien premier avec le produit des éléments déjà sélectionnés. Arrivé à $n/2$ éléments, on teste si $P = M_a^2 + 2$ est un nombre premier. La fonction de test utilisée est simplement la fonction `isprime` de Maple, qui n'effectue qu'un test probabiliste. Il est extrêmement probable que le P trouvé soit premier, mais un test déterministe de primalité devrait être effectué pour une réelle implantation cryptographique sur le P choisi. Générer les P pseudo-premiers est rapide, puisqu'en 15 secondes Maple génère plus de 10 000 P différents de 512 bits, correspondants à 10 000 bases $\mathcal{B}_a$ différentes. Une fois P et la base $\mathcal{B}_a$ déterminés, il ne manque plus qu'à choisir une base $\mathcal{B}_b$ qui soit composée de nombres premiers avec M_a, de sorte à remplir la condition $M_b > 6M_a$ (au minimum). Pour pouvoir utiliser la fonction de compression, on rappelle que la condition est en fait $M_b > 19M_a$, on a donc M_b qui a 5 bits de plus que M_a.

La figure 4.2 présente l'architecture implantée de notre algorithme `SBMM`. Notre implantation utilise ici $n/2$ `Rower`s « habituels », et un petit `Rower` supplémentaire. Le petit `Rower` supplémentaire est en fait dédié à la seconde base, car elle est doit être plus grande que la première comme expliqué précédemment. Dans nos implantations, la base $\mathcal{B}_a$ est constituée de $n/2$ moduli de w bits, et la base $\mathcal{B}_b$ est composée de $n/2$ moduli de w bits et d'un petit modulo de 6 bits. Le petit `Rower` est dédié à ce modulo de 6 bits, que l'on prend égal à 2^6. On rappelle que M_b est plus long d'au moins 5 bits que M_a pour pouvoir utiliser la fonction de compression. Si dans nos résultats d'implantation le modulo supplémentaire est de 6 bits, c'est dû à une estimation de la condition d'utilisation grossière de la fonction de compression que nous avions faite dans un premier temps. La figure reprend les mêmes codes que la figure 2.2 présentée dans le chapitre 2. Les petits cercles sont toujours les signaux de contrôle et les carrés sont toujours la sélection des 6 bits de poids forts parmi les w bits entrant. On a rajouté ici des petits rectangles permettant de passer de 6 bits à w bits, tout simplement en concaténant $w - 6$ zéros devant les 6 bits entrant. Cela permet un contrôle plus simple, car on considère la sortie du petit `Rower` comme celle des autres `Rower`s pour les extensions de base.

Les `Rower`s utilisés sont les mêmes que ceux implantés dans le chapitre 2 pour l'inversion `PM-MI`, et le petit `Rower` supplémentaire lui est très simple, car il calcule simplement des opérations modulo 2^6. Comme les autres `Rower`s, il est doté de 6 étages de pipeline, afin de simplifier le contrôle. Ce modulo supplémentaire n'est pas obligatoire pour obtenir

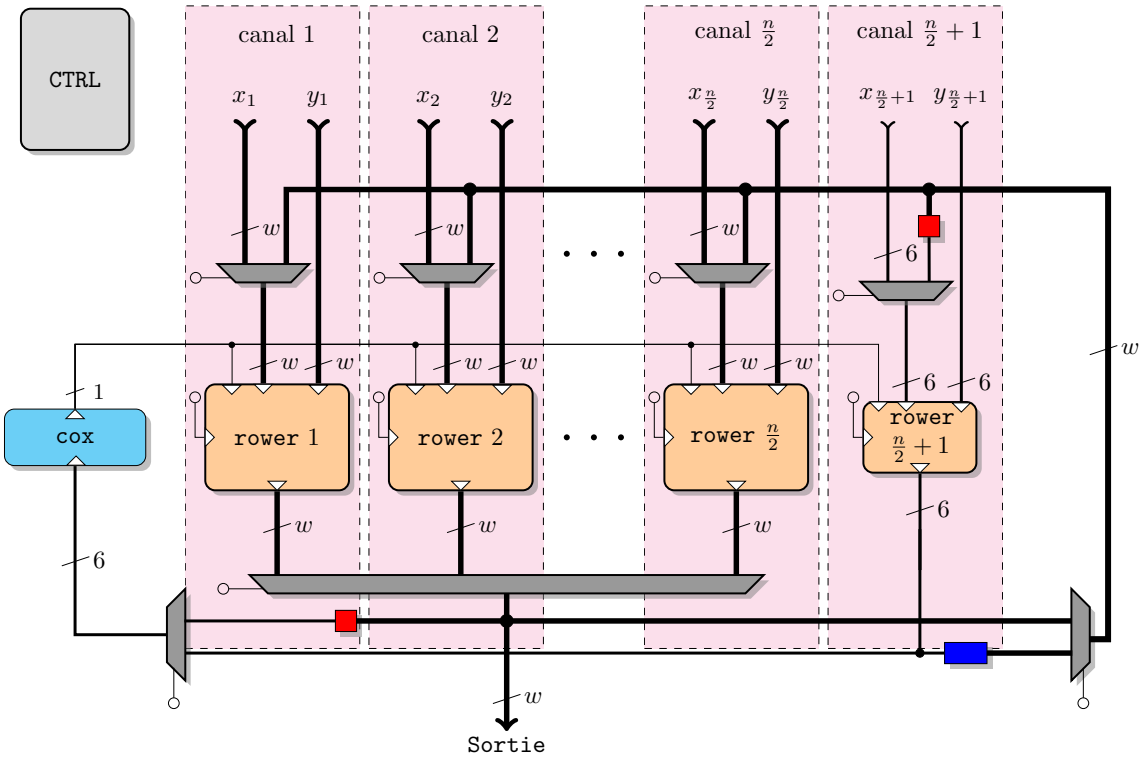

FIGURE 4.2 – Architecture de la multiplication modulaire `SBMM` implantée, avec un modulo de 6 bits supplémentaire pour la base $\mathcal{B}_b$.

M_b plus grand de 5 bits que M_a. En effet, on peut aussi choisir la base $\mathcal{B}_b$ avec des moduli de $w+1$ bits au lieu de w bits. L'avantage d'une telle solution est que, premièrement, elle peut être moins chère en surface que notre implantation si $w+1$ est plus petit ou égal à la taille maximale supportée par les multiplieurs câblés du FPGA. Deuxièmement, le petit `Rower` supplémentaire implique 2 cycles supplémentaires dans notre implantation de l'algorithme `SBMM`, par rapport à cette solution. Mais pour implanter cette solution, il faut réaliser des `Rower`s et un `Cox` qui gèrent 2 tailles d'éléments, w et $w+1$. Par manque de temps, nous avons décidé de garder les `Rower`s qui ne gèrent qu'une seule taille, et d'implanter un petit `Rower` supplémentaire.

Notre implantation ne concerne pour le moment que l'algorithme `SBMM`, sans la fonction de compression. Même si nous avons pris une base $\mathcal{B}_b$ telle que la fonction de compression soit utilisable, notre implantation va juste calculer le résultat d'une multiplication modulaire. Le résultat n'est pas compressé, il est inférieur à $6P$. La compression est censée être effectuée en parallèle sur un petit modulo supplémentaire m_γ. Avec l'architecture illustrée figure 4.2, ces calculs peuvent être effectués sur le `Rower` supplémentaire qui n'est utilisé que la moitié du temps : lorsque l'on calcule sur la base $\mathcal{B}_b$. Il suffira de s'occuper des compressions lorsque les calculs sur les autres `Rower`s seront faits sur la base $\mathcal{B}_a$. Si on implante des `Rower`s gérant aussi des éléments de $w+1$ bits, on est obligés de rajouter un petit `Rower` supplémentaire, qui dans ce cas pourra être utilisé en parallèle des calculs sur $\mathcal{B}_a$, mais aussi en parallèle des calculs sur $\mathcal{B}_b$.

Nous avons choisi d'implanter une architecture sur $n/2$ `Rower`s plutôt que sur n `Rower`s

Algo.	ℓ	$n \times w$	Slices (FF/LUT)	nb. DSP	nb. BRAM	nb. cycles	Fréq. (MHz)	temps (ns)
MM	192	12×17	2011(2956/5906)	26	0	50	208	240
	384	12×33	3304(5692/10455)	84	12	50	118	424
	512	16×33	6180(7557/15240)	112	16	58	116	500
SBMM	192	$12 \times 16^*$	1476(1973/4604)	15	0	58	223	260
	384	$12 \times 32^*$	2256(3818/8415)	42	6	58	124	467
	512	$16 \times 32^*$	3400(4960/10877)	57	8	66	123	535

TABLE 4.4 – Résultats d'implantation des deux algorithmes de multiplication modulaire sur FPGA Virtex 5 (XC5VLX50T pour 192 bits, XC5VLX220 pour 384 et 512 bits). Les temps sont donnés pour une exécution unique. Notation * : un modulo supplémentaire de 6 bits est rajouté à la seconde demi-base $\mathcal{B}_b$.

parce qu'il était plus simple d'adapter nos anciennes implantations à ce cas là. Une première façon d'obtenir une implantation sur n `Rower`s de notre algorithme est de doubler le nombre de total de `Rower`s (y compris le petit `Rower`), et d'effectuer les lignes 2 et 4 de l'algorithme 26 en parallèle. En effet, ces deux opérations sont complètement indépendantes. Les autres opérations de l'algorithme 26 sont effectuées sur les 2 demi-bases $\mathcal{B}_a$ et $\mathcal{B}_b$, donc déjà sur n moduli. Dans tous les cas, ce nouvel algorithme opérant sur des demi-bases, il doit être évalué sur au moins deux architectures, une avec $n/2$ `Rower`s et une avec n `Rower`s.

4.3.2 Résultats d'implantation

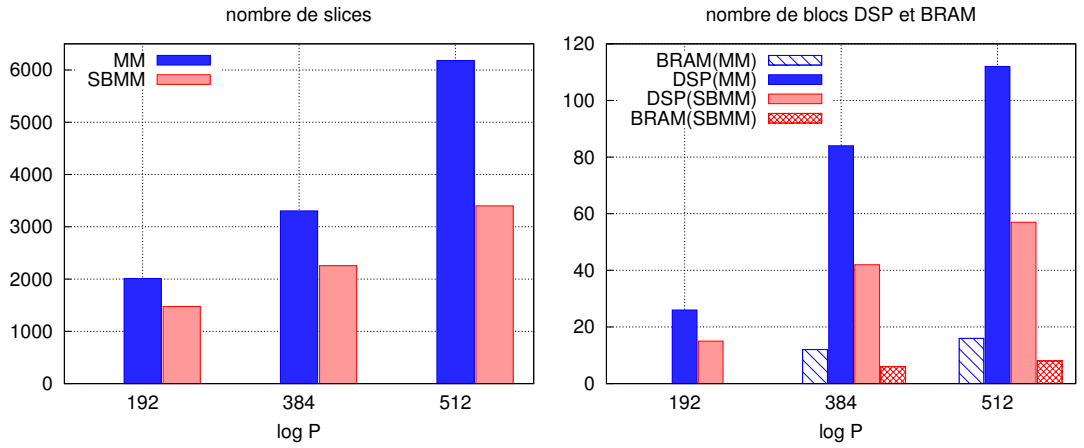

FIGURE 4.3 – Comparaison en surface des implantations FPGA de `MM` et `SBMM`, en nombre de *slices* et en nombre de blocs dédiés (DSP et BRAM).

Les résultats obtenus pour les deux algorithmes sont présentés dans la table 4.4 et sont illustrés par la figure 4.3 pour les résultats de surface et la figure 4.4 pour le temps d'exécution. Les tailles de corps implantées sont 192 et 384 bits, qui sont des tailles de corps standardisées par le NIST [91], et 512 bits. Le standard du NIST propose l'utilisation de courbes définies sur 521 bits car $P = 2^{521} - 1$ est premier, et permet d'avoir une

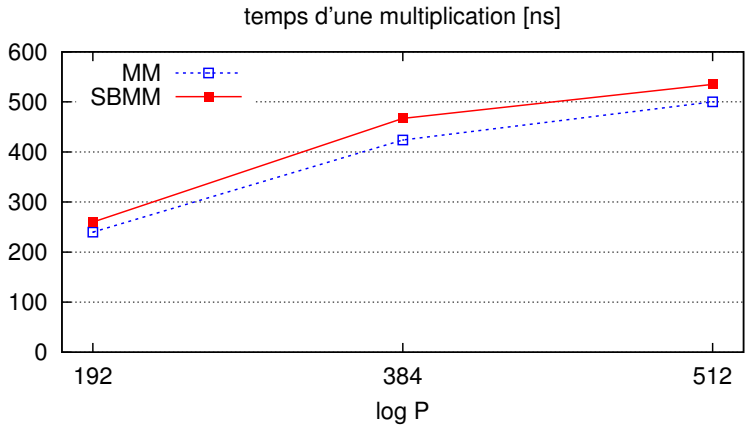

FIGURE 4.4 – Comparaison du temps d'exécution des implantations FPGA de `MM` et `SBMM`.

multiplication modulaire très efficace, en représentation binaire classique. Dans notre cas, nous avons pris un premier de 512 bits car il est facile d'en trouver tels que $P = M_a^2 - 2$. Les mots sont de taille 16 bits pour le corps de 192 bits et 32 bits pour les autres pour l'algorithme `SBMM`, alors que l'on a $w = 17$ et $w = 33$ bits pour l'algorithme `MM` de l'état de l'art. Cela est dû au fait que dans notre implantation, les bits supplémentaires nécessaires sont fournis par le petit `Rower` supplémentaire.

Les résultats présentés à la table 4.4 ont été obtenus sur FPGA Virtex 5, XC5VLX50T pour 192 bits et XC5VLX220 pour 384 et 512 bits. Grâce à notre algorithme, avec l'architecture proposée, nous obtenons une implantation bien plus compacte, pour un faible surcoût en temps. Comme illustré dans la figure 4.3, pour 192, 384 et 512 bits nous obtenons respectivement une réduction du nombre de *slices* de 27 %, 32 % et 45 %. De plus, le nombre de blocs DSP est respectivement réduit de 43 %, 50 % et 50 % et nous avons divisé par 2 le nombre de BRAM pour les 3 corps. En contrepartie, nous obtenons un surcoût en temps d'exécution de 8 %, 10 % et 7 % respectivement pour les corps de 192, 384 et 512 bits, illustré dans la figure 4.4. Comme pour nos propositions des chapitres 2 et 3, nos gains sont plus importants lorsque n devient grand, c'est à dire lorsque nous travaillons sur des hauts niveaux de sécurité. Nous rappelons enfin que l'implantation de notre nouvel algorithme n'a pas encore été optimisée, faute de temps.

4.4 Conclusion

Dans ce chapitre nous avons présenté des travaux introduisant un nouvel algorithme de multiplication modulaire RNS, appelé `SBMM`, utilisable pour la cryptographie sur courbes elliptiques. Grâce à des P bien choisis et à des décompositions semblables à celles présentées dans le chapitre 3 sur le `SPRR`, nous avons divisé par 2 le nombre de multiplications modulaires élémentaires et par 4 le nombre de pré-calculs par rapport à l'algorithme de l'état de l'art. Des premiers résultats d'implantation sur FPGA, partiels, permettent de calculer une multiplication modulaire RNS avec une surface jusqu'à 2 fois plus petite, pour un surcoût en temps d'exécution de seulement 10 % (au plus) pour nos implantations.

Ces travaux étant en cours, des implantations plus poussées seront réalisées, avec notamment l'implantation d'un doublement et d'une addition de points utilisant notre algorithme, associée à une fonction de compression pour pouvoir enchaîner les multiplications `SBMM`. Une implantation orientée vitesse sera aussi réalisée, qui devrait être bien plus rapide que les implantations de l'état de l'art, pour un faible surcoût en surface.

Enfin, ces travaux devront être valorisés via une implantation complète d'une multiplication scalaire, combinés avec les travaux sur l'inversion modulaire présentés dans le chapitre 2.

Chapitre 5

Tests de divisibilité multiples

Dans cette section, issue de notre publication [18], nous présentons un opérateur arithmétique matériel dédié aux tests de divisibilité par plusieurs petites constantes sur des grands entiers (comme des scalaires ECC). Ces grands entiers, de plusieurs centaines de bits, sont représentés en multi-précision. La méthode proposée permet de n'effectuer qu'un très faible nombre de calculs pour chaque mot de la représentation multi-précision. Par exemple, elle permet de tester la divisibilité par $(2^a, 3, 5, 7, 9)$, où $1 \leq a \leq 12$, beaucoup plus efficacement qu'en testant la divisibilité par chacune des petites constantes séparément. La méthode proposée a été implantée et validée sur circuit FPGA. Ce chapitre ne traite pas spécifiquement du RNS, mais de calculs simultanés sur des petits moduli, ou bien de recodage à bases multiples de clés pendant la multiplication scalaire [25], ce qui est complémentaire au reste des travaux de la thèse. Enfin, ce chapitre n'utilise pas les notations utilisées pour le RNS, de nouvelles notations seront définies spécifiquement.

5.1 Introduction

Dans certaines applications particulières, il est nécessaire de pouvoir tester rapidement si un entier est divisible par des constantes comme 2, 3, 5 ou d'autres petits premiers et des petites puissances de ces nombres comme 3^2. Ceci se fait assez facilement, en logiciel et en matériel, pour une seule constante et sur un nombre à tester de taille modérée (de la taille d'un mot machine en logiciel ou de quelques dizaines de bits en circuit). Mais lorsqu'il s'agit d'effectuer ces tests sur de grands entiers de plusieurs centaines de bits ou plus (p. ex. pour des tailles de nombres utilisés en cryptographie asymétrique) et pour plusieurs constantes à la fois, ceci nécessite des calculs élémentaires bien plus nombreux et ainsi des opérateurs coûteux à implanter en matériel. Ceci limite considérablement l'application de méthodes utilisant des tests de divisibilité de grands entiers en matériel.

Par exemple, en cryptographie sur courbes elliptiques, une façon d'accélérer la multiplication scalaire est de faire appel à des algorithmes de recodage en bases multiples. Ces recodages permettent de réduire significativement le nombre total d'opérations à effectuer sur les points de la courbe elliptique considérée. En base double $(2, 3)$, on représente un nombre par une somme de termes de la forme $\pm 2^{e_1} 3^{e_2}$, voir [40] par exemple. Pour des bases multiples comme $(2, 3, 5, 7)$, la représentation se fait via la somme de termes de la forme $\pm 2^{e_1} 3^{e_2} 5^{e_3} 7^{e_4}$, voir [73] par exemple. Notre équipe travaille sur de tels recodages qui nécessitent des tests de divisibilité efficaces par les différents éléments des bases multiples (et si possible des petites puissances de ces premiers).

Dans ce chapitre, nous présentons une méthode permettant d'effectuer simultanément, rapidement, et sur des petits circuits, les tests de divisibilité par plusieurs petites constantes comme $(2^a, 3, 5, 7, 9)$, avec a petit, sur de très grands entiers de plusieurs centaines de bits et représentés en multi-précision (c.-à-d. sous forme de vecteurs de mots de taille modérée). La méthode proposée repose sur l'adaptation d'une très ancienne méthode décrite par Blaise Pascal [93, 103]. Nous l'avons adaptée au cas de la divisibilité par plusieurs constantes et en matériel.

La section 5.2 présente les notations utilisées dans ce chapitre et quelques hypothèses qui seront utiles pour les implantations matérielles. L'état de l'art du domaine est présenté en section 5.3. La section 5.4 présente la version simple en base 2, directement issue de la méthode de Pascal, de l'opérateur de divisibilité et son implantation FPGA. La section 5.5 présente notre amélioration de la méthode en utilisant une base intermédiaire plus grande de type 2^v. Nous donnons aussi les résultats d'implantation FPGA pour cette amélioration. La section 5.6 décrit brièvement des comparaisons avec d'autres travaux proches. Enfin, la section 5.7 présente la conclusion et quelques perspectives.

5.2 Notations et hypothèses d'implantation matérielle

L'*entier naturel* x est l'opérande dont la divisibilité doit être testée. Il est représenté en numération simple de position de base 2 (la notation binaire standard, cf. [85, p. 27] par exemple) sur n bits. On a donc :

$$x \quad = \quad (x_{n-1}x_{n-2}\dots x_1x_0)_2 \quad = \quad \sum_{i=0}^{n-1} x_i 2^i \quad .$$

La notation $()_2$ signifie que les éléments entre parenthèses sont les bits de la représentation de base 2 avec les poids faibles à droite. Dans nos applications cibles — l'implantation matérielle de crypto-processeurs ECC — les nombres ont des tailles dans l'intervalle $n \in [160, 600]$ bits, mais notre méthode fonctionne directement pour des tailles plus importantes. L'extension au cas des entiers relatifs est triviale et ne sera pas décrite dans ce chapitre (car peu utile dans nos applications en cryptographie).

En pratique dans le circuit, un tel grand entier est représenté — et donc stocké — en multi-précision par un *vecteur* de t *mots* de taille w bits. Le nombre t de mots nécessaires pour représenter les entiers de n bits est donné par la relation : $w \cdot (t-1) < n \leq w \cdot t$. Si besoin, le mot de poids le plus fort est complété par des zéros (0-*padding*). On note $x^{(j)}$, avec $0 \leq j < t$, le j-ième mot de la représentation multi-précision de x en partant des poids faibles. Le stockage de x en multi-précision est illustré au niveau architecture en figure 5.1. L'adresse du j-ième mot de x est entrée dans le bloc mémoire et le contenu de ce mot $x^{(j)}$ sort sur le port de lecture. Dans nos implantations matérielles en FPGA, ce bloc mémoire sera implanté en LUT (*look-up table*) afin de pouvoir mesurer l'impact de n et w sur la surface de circuit utilisée. Bien évidement, notre méthode s'applique directement aux implantations avec des blocs dédiés de mémoire câblée des FPGA modernes (p. ex. les BRAM des FPGA Xilinx).

Nous noterons $\mathcal{D}$ l'ensemble des l diviseurs par lesquels on souhaite tester la divisibilité de x, on a ainsi $\mathcal{D} = (d_1, d_2, \dots, d_l)$. Nous désignons par d un des éléments de $\mathcal{D}$ (quand sa position dans $\mathcal{D}$ n'a pas d'importance afin d'alléger les notations). Les diviseurs de $\mathcal{D}$

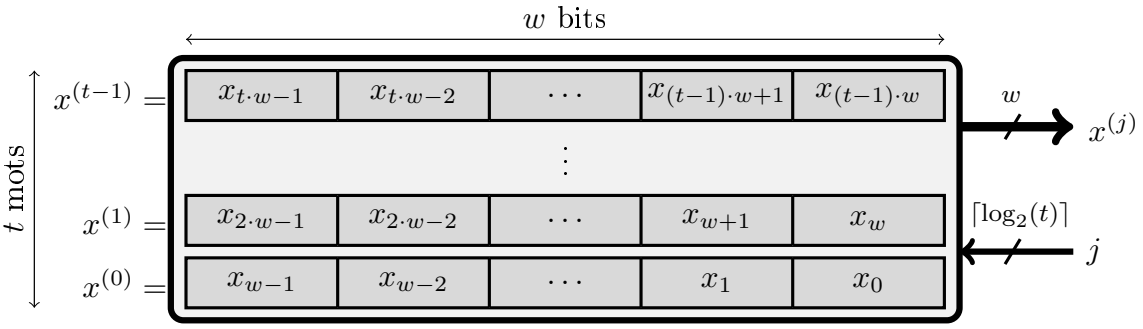

FIGURE 5.1 – Stockage de l'argument x sur t mots de w bits.

envisagés pour les tests sont des petits nombres premiers comme 2, 3, 5 ou 7 ainsi que quelques puissances limitées de ces premiers comme 2^a (avec $a \leq w$ en pratique dans l'architecture) ou 3^2.

Les réalisations matérielles de ce chapitre ont été décrites en VHDL puis implantées sur un FPGA XC5VLX50T en utilisant l'environnement ISE 12.4, tous deux de la société Xilinx, avec des paramètres d'effort standard pour la synthèse et le placement/routage. Dans la suite, nous résumerons les résultats d'implantation, obtenus par les outils, en termes de nombre de cycles d'horloge, de surfaces exprimées en *slices* et de fréquences d'horloge. Les surfaces seront aussi indiquées en nombre de LUT (à 6 entrées dans Virtex 5) et nombre de bascules (FF pour *flip-flop*). Pour information, un XC5VLX50T contient 7 200 *slices* de 4 LUT-6 et 4 FF par *slice*.

5.3 État de l'art

Il existe de nombreuses références qui présentent des tests de divisibilité. Par exemple, les tests élémentaires comme la divisibilité par 3, 5, 9 ou 10 en base 10 se trouvent dans les livres de mathématiques pour le collège. Des listes plus étoffées de tests de divisibilité en bases 10 et 2 se trouvent sur des sites comme Wikipédia. Enfin, des livres bien plus techniques comme [124] proposent des astuces logicielles pour effectuer certains tests très rapidement en utilisant de courtes séquences d'instructions arithmétiques et logiques. Mais il existe très peu de référence sur l'implantation matérielle de ces tests, en particulier pour des grands nombres (cf. section 5.6). Toutes ces méthodes utilisent, plus ou moins directement et avec des adaptations, l'idée de base présentée ci-dessous.

Blaise Pascal (1623–1662) a proposé une méthode générale permettant de tester la divisibilité de x par d dans une représentation de base b quelconque (avec x en numération simple de position, c.-à-d. $x = \sum_{i=0}^{n-1} x_i b^i$). Cette méthode est décrite dans une publication posthume, en latin, de 1819 [93] [1]. Une analyse moderne et en anglais de ce texte se trouve dans [103] (on trouve sur Internet une version équivalente de ce texte et en français sous le titre « La machine à diviser de Monsieur Pascal »). Cette méthode est souvent appelée *ruban de Pascal*. La méthode décrite par Pascal est peut-être encore plus ancienne, mais nous ne connaissons pas de texte antérieur présentant cette méthode en base b quelconque.

Pascal a remarqué que les valeurs des restes $b^i \bmod d$, avec $i \in \mathbb{N}$ et d premier avec la base b, forment une séquence périodique très simple dans certains cas. Dans la suite de ce

1. Nous remercions Christiane Frougny pour nous avoir fourni les références historiques sur le sujet.

chapitre, nous nous limiterons au cas de la base $b = 2$ ou $b = 2^v$ avec v petit. La table 5.1 présente les restes $2^i \bmod d$ pour $d \in \{3, 5, 7\}$ et $i \leq 16$. Chaque ligne de la table 5.1 présente le ruban de Pascal correspondant à d.

	i																
d	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	**2**	**1**
5	1	3	4	2	1	3	4	2	1	3	4	2	1	**3**	**4**	**2**	**1**
7	2	1	4	2	1	4	2	1	4	2	1	4	2	1	**4**	**2**	**1**

TABLE 5.1 – Rubans de Pascal pour la divisibilité par $d \in \{3, 5, 7\}$ (les valeurs sont $2^i \bmod d$).

À partir de la table 5.1, on peut utiliser le ruban de Pascal pour tester facilement la divisibilité par $d = 3$, pour lequel la séquence périodique est $(2\,1)^*$. Ce ruban indique que :

$$\begin{aligned} x \bmod 3 &= (\ldots + 2^5 x_5 + 2^4 x_4 + 2^3 x_3 + 2^2 x_2 + 2^1 x_1 + x_0) \bmod 3 \\ &= (\ldots + 2\,x_5 + \quad x_4 + 2\,x_3 + \quad x_2 + 2\,x_1 + x_0) \bmod 3 \\ &= \left(\underbrace{\sum (2x_{2i+1} + x_{2i})}_{\alpha} \right) \bmod 3 \end{aligned}$$

Cela signifie qu'il faut commencer par sommer les sous-mots de taille 2 bits de type $(x_{2i+1}\, x_{2i})_2$ sur toute la largeur de l'opérande. La somme obtenue est alors congrue à 3 dans le cas où x est divisible par 3. En appliquant récursivement cette méthode sur l'écriture de la somme obtenue pour α (la taille de α dépend de t), comme illustré à la figure 5.2, on obtient une valeur pour laquelle il suffit de tester si elle est égale à 0 ou à 3. Si la valeur finale obtenue est différente de 0 et de 3 alors x n'est pas divisible par $d = 3$.

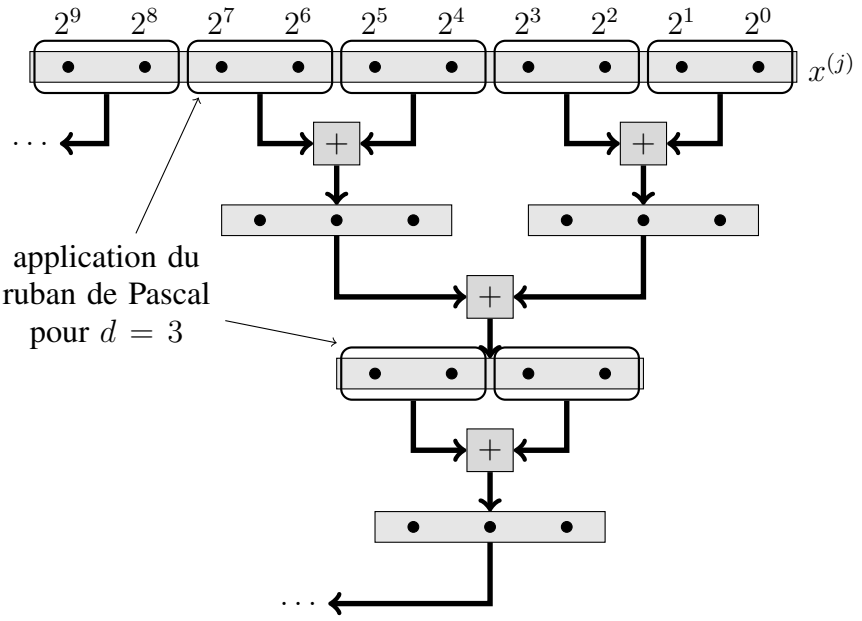

FIGURE 5.2 – Application récursive du ruban de Pascal pour $d = 3$.

Cette méthode s'applique directement au cas $d = 7$ pour lequel la séquence périodique

est $(4\,2\,1)^*$. Ce qui signifie qu'il faut faire la somme α (spécifique à cette valeur de d) des sous-mots de taille 3 bits et de la forme $(x_{3i+2}\,x_{3i+1}\,x_{3i})_2$ le plus possible (en appliquant la décomposition récursivement), puis enfin tester si la somme réduite finale est égale à 0 ou 7.

Mais dans le cas $d = 5$, la séquence périodique est $(3\,4\,2\,1)^*$. On ne peut plus faire la somme par sous-mots de taille 4 bits du fait du facteur 3 (3 n'étant pas une puissance de 2). Pour résoudre ce problème, deux solutions s'offrent à nous : considérer $3 = 1+2$ ou bien considérer $3 \equiv -2 \bmod 5$. La première solution consiste à utiliser le bit de rang $4i+3$ comme entrée de l'addition aux rangs $4i+1$ et $4i$ simultanément. La seconde nécessite d'utiliser la représentation en complément à deux pour effectuer toutes les sommes intermédiaires pour obtenir α.

Le test de divisibilité par 2^a avec $a \in \mathbb{N}$ est trivial pour x représenté en base 2. Il suffit en effet de regarder si les a bits de poids faibles sont nuls ou non.

5.4 Utilisation directe des rubans de Pascal en base 2

Dans un premier temps, nous avons utilisé directement la méthode des rubans de Pascal en base 2, c.-à-d. en considérant les sommes de sous-mots de 2 bits pour $d = 3$ et 3 bits pour $d = 7$ par exemple. L'architecture correspondante est présentée en figure 5.3. Nous avons implanté une version pour $\mathcal{D} = (2^{1\cdots a}, 3, 5, 7)$. Les blocs de somme « $\sum$ » calculent la valeur de α propre à chaque diviseur d de $\mathcal{D}$ différent de $2^{1\cdots a}$. En plus, dans ces blocs, nous appliquons le ruban de Pascal pour chaque d afin de réduire la taille du registre d'accumulation. Donc, nous n'accumulons pas réellement la somme α, mais la somme α après application du ruban de Pascal. Lors de cette accumulation/réduction, il faut traiter les t mots de l'opérande x. Les blocs « $\mathcal{R}$ », quant à eux, effectuent les toutes dernières applications du ruban de Pascal ainsi que le test final (p. ex. tester si on a une valeur égale à 0 ou à d). Les signaux d'horloge et de contrôle ne sont pas représentés sur la figure 5.3.

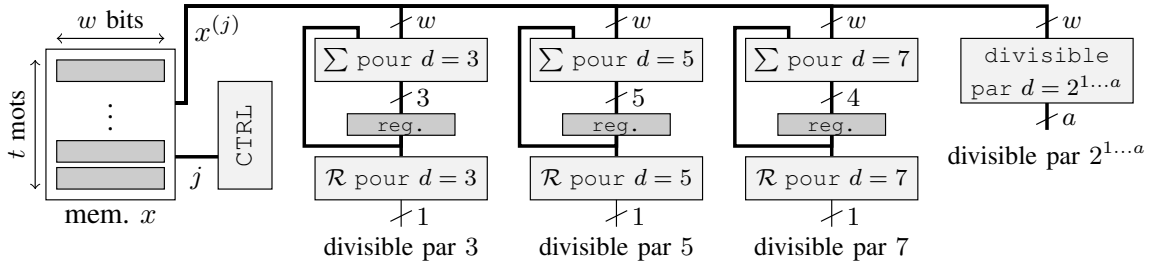

FIGURE 5.3 – Architecture des tests de divisibilité par $\mathcal{D} = (2^{1\cdots a}, 3, 5, 7)$ en utilisant directement les rubans de Pascal en base 2.

Nous avons testé les deux solutions pour le test de divisibilité par $d = 5$: considérer $3 = 1 + 2$ ou bien considérer $3 \equiv -2 \bmod 5$. C'est la version non signée, $3 = 1 + 2$, qui s'avère être la plus efficace en vitesse et en surface. Ceci est très probablement lié au surcoût de l'extension de signe pour les additions/soustractions en complément à deux.

L'architecture de la figure 5.3 permet de tester la divisibilité par chacun des éléments de $\mathcal{D}$ de façon simultanée et en un seul parcours des mots de la représentation de x. Le nombre de cycles d'horloge nécessaire est $t+O(1)$ où la constante dépend de la taille w des mots. Le paramètre w influence grandement les performances et la taille de l'opérateur.

Les longueurs des séquences périodiques de restes $2^i \bmod d$ pour $d = 3, 5, 7$ sont respectivement $2, 4, 3$ (voir table 5.1). Afin d'éviter un décodage complexe, nous utilisons le plus petit commun multiple de ces longueurs pour la valeur de w. Dans notre cas particulier, on a donc $w = \text{ppcm}(2, 4, 3) = 12$. En pratique, on peut utiliser des multiples du ppcm. Nous avons testé $w = 12$ et $w = 24$ (de plus grandes valeurs réduisent beaucoup la fréquence d'horloge et nécessitent l'introduction d'étages de pipeline supplémentaires).

Les résultats d'implantation en FPGA sont résumés dans la table 5.2 pour des opérandes de $n = 160$ bits (la plus petite taille utile pour nos applications ECC) et pour des tests de divisibilité par $\mathcal{D} = (2^{1\cdots a}, 3, 5, 7)$. Nous avons utilisé $a = 12$ quelque soit la valeur w. Ce choix du paramètre a limité à 12 est justifié par des évaluations statistiques sur nos applications ECC.

w	t	surface *slices* (FF/LUT)	fréquence MHz	nombre de cycles
12	14	37 (90/100)	418	$t + 3$
24	7	42 (100/105)	408	$t + 4$

TABLE 5.2 – Résultats d'implantation sur FPGA des tests de divisibilité par $\mathcal{D} = (2^{1\cdots a}, 3, 5, 7)$ utilisant directement les rubans de Pascal en base 2 et pour $n = 160$ bits.

Dans la table 5.2, nous exprimons le nombre de cycles d'horloge nécessaire pour effectuer les tests de divisibilité simultanés en fonction de t (le nombre de mots de x). En effet, il faut lire chaque mot pour accumuler sa contribution à α. La valeur de t utilisée est déterminée par les paramètres n et w (n pour l'application et w pour l'architecture). On rappelle que $w \cdot (t-1) < n \leq w \cdot t$. La fréquence de l'opérateur, quant à elle, dépend fortement des paramètres t et w (et donc indirectement aussi de n).

Notre méthode fonctionne sans aucun problème pour des tailles plus importantes que $n = 160$. Le surcoût en surface de circuit est alors celui du stockage d'arguments de plus grande taille. Ces plus grands arguments nécessitent un compteur de boucle un peu plus large (la taille du compteur étant en $\lceil \log_2(t) \rceil$). Enfin, nous avons validé fonctionnellement nos opérateurs par des simulations aléatoires intensives.

5.5 Amélioration via les rubans de Pascal en grande base 2^v

Appliquer directement la méthode des rubans de Pascal à des tests de divisibilité plus nombreux, comme $\mathcal{D} = (2^{1\cdots a}, 3, 5, 7, 9, 11, 13)$, complique sensiblement le contrôle et augmente le nombre de registres internes utilisés pendant la boucle d'accumulation. La table 5.3 fournit les rubans de Pascal pour $d \in \{9, 11, 13\}$ à partir des bits de x (c.-à-d. les restes sont les valeurs $2^i \bmod d$). Elle montre clairement qu'avec des valeurs de d plus grandes, la longueur des séquences périodiques et leur « complexité » (forme des coefficients) croît de façon importante. Devoir organiser le décodage de ce qu'il convient de faire de chaque bit en fonction du ppcm de ces longueurs va engendrer un contrôle bien plus complexe. De plus, il faut accumuler les valeurs α propres à chaque diviseur d dans des registres distincts.

Afin d'étudier les liens entre les longueurs et la forme des séquences périodiques dans les rubans de Pascal d'une part, et les performances et le coût des opérateurs nécessaires pour des tests plus complexes d'autre part, nous avons modifié différents paramètres arithmétiques et de l'architecture.

	i																			
d	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
9	2	1	5	7	8	4	2	1	5	7	8	4	2	1	**5**	**7**	**8**	**4**	**2**	**1**
11	6	3	7	9	10	5	8	4	2	1	**6**	**3**	**7**	**9**	**10**	**5**	**8**	**4**	**2**	**1**
13	11	12	6	3	8	4	2	1	**7**	**10**	**5**	**9**	**11**	**12**	**6**	**3**	**8**	**4**	**2**	**1**

TABLE 5.3 – Rubans de Pascal en base 2 pour $d \in \{9, 11, 13\}$ (les valeurs sont $2^i \bmod d$).

En faisant varier ces paramètres, nous avons remarqué une propriété très intéressante si l'on considère les restes de sous-mots au lieu des restes des bits (au sens arithmétique) directement. Supposons que l'on découpe les mots de w bits en sous-mots de v bits avec $v \leq w$, c.-à-d. que l'on lit les chiffres de x en base $b = 2^v$. En considérant les restes $\left(2^v\right)^i \bmod d$ au lieu de $2^i \bmod d$, on obtient des séquences périodiques très intéressantes pour certaines valeurs de v et de d. En particulier pour $v = 12$, on obtient les rubans de Pascal présentés à la table 5.4 pour $d \in \{3, 5, 7, 9, 11, 13, 16, 25\}$. Dans cette table, les valeurs sont les restes $\left(2^{12}\right)^i \bmod d$ pour $0 \leq i \leq 9$.

	i									
d	9	8	7	6	5	4	3	2	1	0
3	1	1	1	1	1	1	1	1	1	**1**
5	1	1	1	1	1	1	1	1	1	**1**
7	1	1	1	1	1	1	1	1	1	**1**
9	1	1	1	1	1	1	1	1	1	**1**
11	3	9	5	4	1	**3**	**9**	**5**	**4**	**1**
13	1	1	1	1	1	1	1	1	1	**1**
17	16	1	16	1	16	1	16	1	**16**	**1**
25	6	11	16	21	1	**6**	**11**	**16**	**21**	**1**

TABLE 5.4 – Rubans de Pascal en base 2^{12} pour $d \in \{3, 5, 7, 9, 11, 13, 16, 25\}$ (les valeurs sont $\left(2^{12}\right)^i \bmod d$).

Les lignes qui correspondent aux diviseurs 3, 5, 7, 9 et 13 de la table 5.4 présentent la même séquence périodique $(1)^*$. Ces lignes signifient que les tests correspondants peuvent être effectués en utilisant un unique registre d'accumulation. Le même accumulateur est alors partagé pendant les t cycles de l'accumulation, puis il faut effectuer une réduction finale propre à chaque diviseur dans notre nouvelle architecture illustrée en figure 5.4. L'optimisation provient de l'accumulation d'une seule valeur α partagée par tous les diviseurs ayant $(1)^*$ comme séquence périodique. Afin de tirer parti de cette propriété, il convient de choisir pour w un multiple de v (c.-à-d. $w = v$, $w = 2 \cdot v$, $w = 3 \cdot v$, ...). Dans notre cas, nous avons implanté la version améliorée pour $v = 12$ et $w \in \{12, 24\}$. Dans cette version, la sortie du registre d'accumulation est plus grande ($w + \lceil \log_2(t) \rceil$ bits) que les sorties des registres de la figure 5.3. Les blocs de réduction « $\mathcal{R}'$ » propres à chaque diviseur utilisent la technique spécifique de base $b = 2$ présentée en section 5.3 (figure 5.2).

Les résultats d'implantation en FPGA pour notre méthode améliorée sont donnés dans la table 5.5 pour des opérandes de $n = 160$, 256 et 521 bits (qui correspondent à des tailles cryptographiques assez classiques pour ECC) et pour des divisibilités par $\mathcal{D} =$

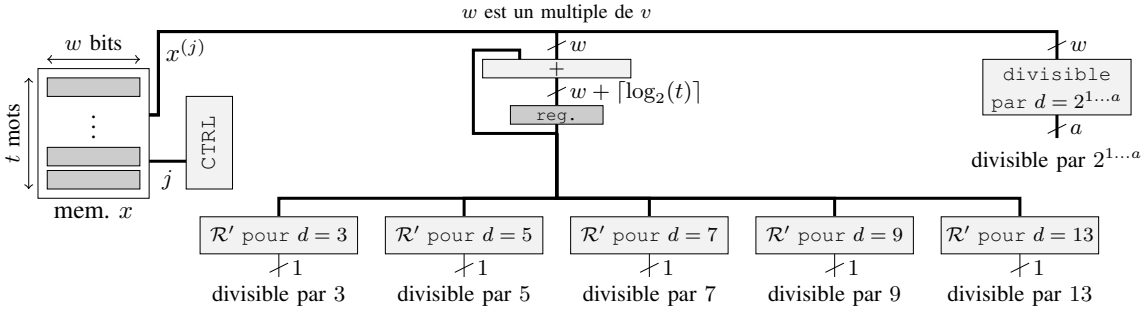

FIGURE 5.4 – Architecture des tests de divisibilité par $\mathcal{D} = (2^{1...a}, 3, 5, 7, 9, 13)$ en utilisant l'amélioration en base 2^v des rubans de Pascal.

$(2^{1...a}, 3, 5, 7, 9, 13)$. Ici aussi, nous avons utilisé $a = 12$ pour les différentes valeurs de w.

n	w	t	surface *slices* (FF/LUT)	fréquence MHz	nombre de cycles
160	12	14	49 (112/135)	454	$t+3$
	24	7	72 (176/198)	490	$t+4$
256	12	22	54 (127/149)	460	$t+3$
	24	11	74 (188/205)	495	$t+4$
521	12	44	56 (129/155)	424	$t+3$
	24	22	74 (192/208)	485	$t+4$

TABLE 5.5 – Résultats d'implantation sur FPGA des tests de divisibilité par $\mathcal{D} = (2^{1...a}, 3, 5, 7, 9, 13)$ utilisant l'amélioration des rubans de Pascal en base 2^{12} et pour $n \in \{160, 256, 521\}$ bits.

5.6 Comparaisons

On trouve dans [47] une application du RNS au test de divisibilité en matériel mais sans aucun résultat d'implantation. La méthode décrite dans [47] suppose que l'opérande x soit représenté en RNS ce qui n'a pas vraiment de sens pour les applications envisagées en cryptographie ECC. On peut représenter les coordonnées des points d'une courbe elliptique (éléments d'un corps fini) avantageusement en RNS, comme le montrent les premiers chapitres de ce document. Mais nous ne connaissons pas de proposition de cryptosystème dans lequel le scalaire k est représenté en RNS pour l'opération de multiplication scalaire $[k]P$.

On trouve sur Internet le texte correspondant à un poster [101] prétendument publié à la conférence FCCM (*Field-Programmable Custom Computing Machines*) 2002 mais ce papier n'est pas dans la liste des posters ou papiers présentés à la conférence (ni pour les autres années à FCCM, ni sur DBLP et IEEEXplore qui stockent pourtant toutes les publications à FCCM, ni dans une autre conférence). Ce texte présente, sans aucun détail, des résultats d'implantation en FPGA de tests de divisibilité pour des nombres en multi-précision avec un temps de calcul qui semble quadratique (en tous cas non-linéaire). Notre méthode a un temps de calcul seulement linéaire en $t + O(1)$ cycles et où le terme constant

est tout petit (3 ou 4 selon les versions).

Nous ne connaissons pas d'autre référence qui traite d'implantation matérielle de tests de divisibilité pour des grands entiers et qui en détaille les performances. Ceci est probablement lié au fait que l'utilisation de tels tests de divisibilité sur des grands nombres en matériel est rarissime. Nous espérons que ce travail permettra d'utiliser ces tests de divisibilité du fait de leur assez bonne efficacité.

5.7 Conclusion

Nous avons proposé une architecture d'opérateur arithmétique matériel dédié aux tests de divisibilité par des petites constantes pour de grands entiers. La méthode proposée permet d'effectuer simultanément différents tests comme les divisibilités par $(2^{1\cdots a}, 3, 5, 7, 9, 13)$ avec $a \leq 12$ en une seule lecture des chiffres de l'opérande à tester. Les résultats d'implantation FPGA montrent que ces tests s'effectuent très rapidement (tant au niveau de la fréquence de l'opérateur que du nombre de cycles d'horloge nécessaire) et enfin sur de petits circuits.

Dans l'avenir, nous souhaitons pouvoir évaluer l'utilisation d'autres propriétés arithmétiques pour permettre les tests de divisibilité par encore plus de diviseurs.

Conclusion

Pour conclure ce document de thèse, nous allons rappeler les différentes contributions proposées puis présenter certaines perspectives pour des travaux futurs sur le RNS.

Tout d'abord, est présenté au chapitre 2, un nouvel algorithme d'inversion modulaire en RNS, appelé `PM-MI`, basé sur l'algorithme d'Euclide étendu classique, et une variante « binaire-ternaire ». Nous avons obtenu de très bons résultats, aussi bien théoriques que d'implantation sur FPGA. Notre algorithme `PM-MI` divise de 12 à 37 fois le nombre de multiplications modulaires élémentaires `EMM` par rapport à l'inversion RNS de l'état de l'art, et sa variante binaire-ternaire de 18 à 54 fois, pour les paramètres (standards) testés. De plus, l'implantation du `PM-MI` pour 4 tailles standardisées par le NIST [91] donne des inversions 5 à 12 fois plus rapides sur FPGA Virtex 5 que celle de l'état de l'art, avec un surcoût en surface très faible. Notre algorithme est d'ailleurs désavantagé par l'architecture, car nous nous sommes fixés comme contrainte de garder l'architecture ECC en RNS de l'état de l'art, et de n'apporter que de petites modifications. En effet, l'architecture doit en premier lieu être adaptée aux calculs pour ECC plutôt que pour l'inversion, car c'est la multiplication scalaire $[k]\mathbf{P}$ d'ECC qui prend la majorité du temps. Notre algorithme utilise bien plus d'additions que de multiplications, à la différence de l'algorithme de l'état de l'art. L'architecture implantée favorisant les multiplications au profit des additions, le temps d'exécution de notre algorithme pourrait être amélioré sur une architecture mieux adaptée. Deux pistes dans la lignée de ces travaux sont à exploiter pour les compléter. Premièrement, il faudra faire une implantation complète d'une multiplication scalaire, avec cette inversion pour mieux mesurer son impact en terme de surface sur la totalité du cryptoprocesseur ECC. Deuxièmement, les améliorations théoriques apportées par la variante binaire-ternaire devront être complétées via une implantation matérielle, qui sera comparée à l'implantation du `PM-MI` binaire.

Le chapitre 3, lui, présente deux contributions, avec certaines idées en commun. La première contribution est un algorithme de multiplication modulaire en RNS, nommé `SPRR`, qui via des décompositions permet de tirer parti des réutilisations d'opérandes. Il permet, par exemple, de réduire le coût d'un carré par rapport à l'algorithme de l'état de l'art, mais nécessite que le corps dans lequel on travaille ait une forme particulière. Cette contrainte est compatible avec les exponentiations pour le logarithme discret, on obtient pour celles-ci jusqu'à 10 % de réduction du nombre de `EMM`, et un nombre de pré-calculs réduit de 25 %. Pour ECC, les résultats obtenus dépendent fortement de la réutilisation d'opérandes que l'on retrouve dans les formules. Les résultats sont globalement moins bons que pour les exponentiations. Nous avons testé trois jeux de formules d'opérations sur les points, pour différents choix de courbes et de coordonnées. Le nombre de pré-calculs est toujours réduit de 25 % environ, mais le nombre de multiplications modulaires élémentaires `EMM` est plus élevé de 5 à 20 % pour 2 des jeux de formules testés. Par contre, Le dernier jeu de

formules permet de réduire jusqu'à 10 % le nombre d'`EMM` pour des grands paramètres ECC. Nous souhaitons étudier dans l'avenir l'impact de cet algorithme sur les possibilités d'aléa qu'offre le RNS pour protéger le circuit. En effet, une partie des moduli ($n/2$ précisément), sont fixés car ils sont liés au corps sur lequel on calcule. Ensuite, nous souhaitons implanter en FPGA l'algorithme afin de tester le réel impact de nos métriques dans un cas concret. Pour les exponentiations nécessaires au logarithme discret, nous avons réduit le nombre d'opérations et le nombre de pré-calculs, nous nous attendons à des résultats d'implantation un peu meilleurs que l'état de l'art.

La seconde contribution du chapitre 3 est un nouvel algorithme d'exponentiation RNS, qui, cette fois, n'impose aucune contrainte sur le corps (ou l'anneau) sur lequel on calcule. Cet algorithme est utilisable pour RSA, à la différence du `SPRR`. On améliore le nombre d'`EMM` de plus de 15 % pour un algorithme de type carrés et multiplications par rapport à l'algorithme d'exponentiation RNS de l'état de l'art, et de 22 % pour un algorithme protégé SPA de type « échelle de Montgomery ». De plus, cet algorithme, à la différence du `SPRR`, est complètement compatible avec les protections basées sur le RNS du type LRA [11]. En contrepartie, on a un surcoût en mémoire de 50 %, cet algorithme est destiné à des implantations focalisées sur la vitesse d'exécution. Comme pour le `SPRR`, cette exponentiation devra être testée en matériel pour juger réellement du compromis temps surface qu'elle offre.

La dernière contribution sur la représentation RNS et son utilisation pour ECC est présentée au chapitre 4. Ce chapitre traite d'un nouvel algorithme de multiplication modulaire en RNS, le `SBMM`, utilisable pour ECC mais pas pour les exponentiations RSA, ni pour celles du logarithme discret. Dans ce chapitre sont définis des nombres premiers P, qui ont une forme très particulière, et qui sont liés à une base RNS. Une des idées de ce chapitre est de générer des premiers P, qui sont au RNS ce que les nombres premiers pseudo-Mersenne sont à la représentation binaire classique. Notre algorithme de multiplication `SBMM` tire parti de cette forme particulière. On obtient un algorithme qui requiert 2 fois moins d'`EMM`, qui divise par 4 le nombre de pré-calculs, et qui requiert 2 fois moins de moduli que l'état de l'art. Des premiers résultats d'implantation sont fournis, avec des gains en surface de 30 à 50 % (en *slices*, blocs DSP et BRAM), pour un surcoût en temps de moins de 10 % (notre nouvelle implantation n'étant pas encore optimisée). Par contre, notre algorithme retourne des sorties un peu plus grandes que les entrées, ce qui contraint à utiliser une fonction de compression. Des solutions ont été proposées pour celle-ci, ne demandant *a priori* que peu de matériel, et pouvant être effectuées en parallèle des autres calculs. Il manque à implanter ces compressions, et à implanter une séquence d'opérations complète correspondant à une multiplication scalaire, pour étudier l'impact de celles-ci. De plus, une implantation orientée vitesse devra aussi être réalisée, où l'on espère diviser par 2 (ou presque), le temps d'exécution, pour un surcoût en surface très faible.

Le chapitre 5 est consacré à une contribution à part dans les travaux de thèse, puisqu'il ne traite pas du RNS. Ce chapitre présente une proposition architecturale qui permet de factoriser certains calculs, lorsqu'on effectue des tests de divisibilité simultanés, par des petites constantes, sur de grands entiers (typiquement des tailles cryptographiques). Ces petites constantes sont soit des nombres premiers, soit des petites puissances de nombre premier, pour une application de notre opérateur à un recodage multi-bases, par exemple dans les travaux [25] effectués dans l'équipe. Un exemple d'ensemble de diviseurs est $2^a, 3, 5, 7, 9$. Notre proposition permet notamment, à moindre coût, d'avoir plus de tests de divisibilité, avec des divisibilités plus compliquées à tester, grâce à l'étude des rubans de Pascal de ces diviseurs pour des grandes bases 2^w. Les premiers résultats FPGA sont intéressants, car ils

donnent des opérateurs petits et très rapides, ce qui permet leur utilisation efficace pour un recodage multi-bases, en parallèle des calculs de la multiplication scalaire. Ces travaux devront être étendus, en particulier en définissant un modèle du coût de l'implantation matérielle sur FPGA de ces multi-tests, pour un ensemble de diviseurs donné. Une étude plus poussée avec beaucoup d'implantations devra être effectuée.

Pour conclure, les travaux de cette thèse ont permis au RNS de bénéficier de nouvelles optimisations, analogues à celles que l'on trouvait en représentation classique, mais qui n'existait pas jusqu'à maintenant en RNS. La représentation RNS bénéficiait déjà d'une adaptation optimisée de l'algorithme de réduction de Montgomery. Nos travaux permettent de profiter aussi, maintenant, d'une adaptation de l'algorithme d'Euclide étendu, d'un équivalent aux multiplications modulo un nombre pseudo-Mersenne, et d'un carré modulaire moins cher qu'une multiplication modulaire quelconque. Certains de ces nouveaux outils restreignant l'utilisation des contre-mesures basées sur le RNS, du type LRA [11], on se retrouve face à deux voies possibles pour le RNS. D'un côté, une utilisation du RNS en tant que contre-mesure, de l'autre côté une utilisation du RNS en tant qu'arithmétique à hautes performances. Dans tous les cas, un premier travail qui pourrait faire suite à ces travaux est l'étude de nos nouveaux algorithmes, dans un cadre plus étendu que celui que nous nous sommes imposés. En effet, nous sommes restés sur la base de l'architecture `Cox-Rower` de l'état de l'art, permettant une extension de base de Kawamura *et al.* [64] efficace. Peut-être que pour mieux profiter de nos propositions, des nouveaux `Rower`s, une nouvelle architecture ou même d'autres choix pour les extensions de base seraient plus judicieux. Après un certain nombre d'améliorations, il est parfois nécessaire de remettre certaines choses à plat, c'est peut-être ce qui attend les implantations RNS. De plus, il sera important dans l'avenir de donner, à un moment donné, une description précise de quelques configurations complètes de paramètres pour ECC en RNS (corps, base RNS, extension de base et tout autre algorithme utilisé), afin de faciliter l'implantation du RNS pour ceux qui voudraient l'utiliser. En effet, le RNS permettait déjà d'avoir des implantations cryptographiques performantes. Avec les nouvelles propositions de cette thèse et les futurs travaux de recherche, il est possible que la représentation devienne si efficace qu'une standardisation des paramètres ECC favorisant le RNS devienne intéressante. La recherche autour du RNS étant actuellement active, et ses possibilités d'évolutions très nombreuses, le temps n'est sûrement pas encore venu pour une telle standardisation. Peut-être que dans quelques années, l'arithmétique RNS pour la cryptographie sera suffisamment mature pour franchir le cap de paramètres lui étant dédiés dans les standards.

Bibliographie personnelle

Publications

K. Bigou et A. Tisserand. **RNS modular multiplication through reduced base extensions**. Dans *Proc. 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 57–62. IEEE, juin 2014.

K. Bigou et A. Tisserand. **Improving modular inversion in RNS using the plus-minus method**. Dans *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, volume 8086 of LNCS, pages 233–249. Springer, août 2013.

K. Bigou, T. Chabrier, et A. Tisserand. **Opérateur matériel de tests de divisibilité par des petites constantes sur de très grands entiers**. Dans *Actes ComPAS'13 / SympA'15 - Symposium en Architectures nouvelles de machines*, janvier 2013.

Exposés

IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP, juin 2014) :
RNS modular multiplication through reduced base extensions.

Conférence d'Informatique en Parallélisme, Architecture et Système (ComPAS, avril 2014) *Track meilleures contributions scientifiques du GDR ASR* :
Improving modular inversion in RNS using the plus-minus method.

Rencontres Arithmétiques de l'Informatique Mathématique (RAIM, novembre 2013) :
Inversion modulaire RNS sur FPGA.

Workshop on Cryptographic Hardware and Embedded Systems (CHES, août 2013) :
Improving modular inversion in RNS using the plus-minus method.

Crypto'Puces (avril 2013) :
Avancées sur l'utilisation de la représentation RNS pour la cryptographie sur courbes elliptiques.

Journées Codage et Cryptographie (C2, octobre 2012) :
Cryptographie sur courbes elliptiques en représentation modulaire des nombres (RNS).

Journées Thèses DGA-MI (septembre 2012) :
Use of Residue Number System for ECC.

Bibliographie

[1] *Stratix III Device Handbook*, volume 1. March 2010.

[2] *Xilinx UG193 Virtex-5 FPGA XtremeDSP Design Considerations v3.5.* January 2012.

[3] L. Adleman. A subexponential algorithm for the discrete logarithm problem with applications to cryptography. In *Proc. 20th Annual Symposium on Foundations of Computer Science (SFCS)*, pages 55–60, October 1979.

[4] D. Agrawal, B. Archambeault, J.R. Rao, and P. Rohatgi. The EM side-channel(s). In *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523 of *LNCS*, pages 29–45. Springer, August 2002.

[5] S. Antao, J.-C. Bajard, and L. Sousa. RNS-based elliptic curve point multiplication for massive parallel architectures. *The Computer Journal*, 55(5):629–647, May 2012.

[6] J.-C. Bajard, L.-S. Didier, and P. Kornerup. An RNS Montgomery modular multiplication algorithm. *IEEE Transactions on Computers*, 47(7):766–776, July 1998.

[7] J.-C. Bajard, L.-S. Didier, and P. Kornerup. Modular multiplication and base extensions in residue number systems. In *Proc. 15th Symposium on Computer Arithmetic (ARITH)*, pages 59–65. IEEE, April 2001.

[8] J.-C. Bajard, S. Duquesne, and M. D. Ercegovac. Combining leak-resistant arithmetic for elliptic curves defined over Fp and RNS representation. Technical Report 311, IACR Cryptology ePrint Archive, May 2010.

[9] J.-C. Bajard, J. Eynard, and F. Gandino. Fault detection in RNS Montgomery modular multiplication. In *Proc. 21th Symposium on Computer Arithmetic (ARITH)*, pages 119–126. IEEE, April 2013.

[10] J.-C. Bajard and L. Imbert. A full RNS implementation of RSA. *IEEE Transactions on Computers*, 53(6):769–774, June 2004.

[11] J.-C. Bajard, L. Imbert, P.-Y. Liardet, and Y. Teglia. Leak resistant arithmetic. In *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, volume 3156 of *LNCS*, pages 62–75. Springer, 2004.

[12] J.-C. Bajard, M. Kaihara, and T. Plantard. Selected RNS bases for modular multiplication. In *Proc. 19th Symposium on Computer Arithmetic (ARITH)*, pages 25–32. IEEE, June 2009.

[13] J.-C. Bajard, N. Meloni, and T. Plantard. Study of modular inversion in RNS. In *Proc. Advanced Signal Processing Algorithms, Architectures, and Implementations XV*, volume 5910, pages 247–255. SPIE, July 2005.

[14] J.-C. Bajard and N. Merkiche. Double level montgomery Cox-Rower architecture, new bounds. Technical Report 440, IACR Cryptology ePrint Archive, June 2014.

[15] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Proc. 6th International Cryptology Conference (CRYPTO)*, volume 263 of *LNCS*, pages 311–323. Springer, 1986.

[16] D. J. Bernstein and T. Lange. Inverted edwards coordinates. In *Proc. 17th International Symposium on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC)*, volume 4851 of *LNCS*, pages 20–27. Springer, December 2007.

[17] D. J. Bernstein and T. Lange. Explicit-formulas database. `http://www.hyperelliptic.org/EFD`, 2014. April update.

[18] K. Bigou, T. Chabrier, and A. Tisserand. Opérateur matériel de tests de divisibilité par des petites constantes sur de très grands entiers. In *Proc. ComPAS'13 / SympA'15 - Symposium en Architectures nouvelles de machines*, January 2013.

[19] K. Bigou and A. Tisserand. Improving modular inversion in RNS using the plus-minus method. In *Proc. 15th Cryptographic Hardware and Embedded Systems (CHES)*, volume 8086 of *LNCS*, pages 233–249. Springer, August 2013.

[20] K. Bigou and A. Tisserand. RNS modular multiplication through reduced base extensions. In *Proc. 25th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 57–62. IEEE, June 2014.

[21] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *Proc. 16th International Conference on the Theory and Application of Cryptographic (EUROCRYPT)*, volume 1233 of *LNCS*, pages 37–51. Springer, May 1997.

[22] R. P. Brent and H. T. Kung. Systolic VLSI arrays for polynomial GCD computation. *IEEE Transactions on Computers*, C-33(8):731–736, August 1984.

[23] E. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In *Proc. 5th International Workshop on Practice and Theory in Public Key Cryptosystems (PKC)*, volume 2274 of *LNCS*, pages 335–345. Springer, 2002.

[24] Certicom Research. Certicom ECC challenge, 2009.

[25] T. Chabrier and A Tisserand. On-the-fly multi-base recoding for ECC scalar multiplication without pre-computations. In *Proc. 21th Symposium on Computer Arithmetic (ARITH)*, pages 219–228. IEEE, April 2013.

[26] P. W. Cheney. A digital correlator based on the residue number system. *IRE Transactions on Electronic Computers*, EC-10(1):63–70, March 1961.

[27] R. C. C. Cheung, S. Duquesne, J. Fan, N. Guillermin, I. Verbauwhede, and G. X. Yao. FPGA implementation of pairings using residue number system and lazy reduction. In *Proc. 13th Cryptographic Hardware and Embedded Systems (CHES)*, volume 6917 of *LNCS*, pages 421–441. Springer, September 2011.

[28] D. V. Chudnovsky and G. V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics*, 7(4):385–434, 1986.

[29] M. Ciet, M. Neve, E. Peeters, and J.-J. Quisquater. Parallel FPGA implementation of RSA with residue number systems - can side-channel threats be avoided? In *Proc. 46th Midwest Symposium on Circuits and Systems (MWSCAS)*, volume 2, pages 806–810. IEEE, December 2003.

[30] C. C. Cocks. A note on 'non-secret encryption'. *CESG Memo*, 1973.

[31] H. Cohen and G. Frey, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2005.

[32] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *Proc. 4th International Conference on the Theory and Application on Cryptology and Information Security (ASIACRYPT)*, volume 1514 of *LNCS*, pages 51–65. Springer, October 1998.

[33] J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Proc. 1st Cryptographic Hardware and Embedded Systems (CHES)*, volume 1717 of *LNCS*, pages 292–302. Springer, August 1999.

[34] R. E. Crandall. Method and apparatus for public key exchange in a cryptographic system, October 1992. US Patent 5,159,632.

[35] P. de Fermat. *Œuvres de Fermat, tome II, Correspondance*. Gauthier-Villars, 1894.

[36] J.-P. Deschamps and J. L. Imana an G. D. Sutter. *Hardware Implementation of Finite-Field Arithmetic*. McGraw-Hill, 2009.

[37] J.-P. Deschamps and G. Sutter. Hardware implementation of finite-field division. *Acta Applicandae Mathematicae*, 93(1-3):119–147, September 2006.

[38] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[39] V. Dimitrov, L. Imbert, and P. K. Mishra. Efficient and secure elliptic curve point multiplication using double-base chains. In *Proc. 11th International Conference on the Theory and Application on Cryptology and Information Security (ASIACRYPT)*, volume 3788 of *LNCS*, pages 59–78. Springer, December 2005.

[40] V. Dimitrov, L. Imbert, and P. K. Mishra. The double-base number system and its application to elliptic curve cryptography. *Mathematics of Computation*, 77(262):1075–1104, April 2008.

[41] J. D. Dixon. The number of steps in the Euclidean algorithm. *Journal of number theory*, 2(4):414–422, 1970.

[42] S. Duquesne. RNS arithmetic in $\mathbb{F}_p^k$ and application to fast pairing computation. *Journal of Mathematical Cryptology*, 5:51–88, June 2011.

[43] H. M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, July 2007.

[44] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[45] M. Esmaeildoust, D. Schinianakis, H. Javashi, T. Stouraitis, and K. Navi. Efficient RNS implementation of elliptic curve point multiplication over GF(p). *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(8):1545–1549, August 2013.

[46] Euclide. *Les quinze livres des éléments géométriques d'Euclide*. D. Henrion, 1632.

[47] D. Gamberger. Incompletely specified numbers in the residue number system-definition and applications. In *Proc. 9th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 210–215. IEEE Computer Society, September 1999.

[48] F. Gandino, F. Lamberti, G. Paravati, J.-C. Bajard, and P. Montuschi. An algorithmic and architectural study on Montgomery exponentiation in RNS. *IEEE Transactions on Computers*, 61(8):1071–1083, August 2012.

[49] H. L. Garner. The residue number system. *IRE Transactions on Electronic Computers*, EC-8(2):140–147, June 1959.

[50] C. Giraud. An RSA implementation resistant to fault attacks and to simple power analysis. *IEEE Transactions on Computers*, 55(9):1116–1120, September 2006.

[51] D. M. Gordon. A survey of fast exponentiation methods. *Journal of algorithms*, 27(1):129–146, 1998.

[52] N. Guillermin. A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In *Proc. 12th Cryptographic Hardware and Embedded Systems (CHES)*, volume 6225 of *LNCS*, pages 48–64. Springer, August 2010.

[53] N. Guillermin. A coprocessor for secure and high speed modular arithmetic. Technical Report 354, IACR Cryptology ePrint Archive, 2011.

[54] N. Guillermin. *Implémentation matérielle de coprocesseurs haute performance pour la cryptographie asymétrique*. PhD thesis, Université Rennes 1, January 2012.

[55] N. Gura, A. Patel, A.S. Wander, H. Eberle, and S.C. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Proc. 6th Cryptographic Hardware and Embedded Systems (CHES)*, volume 3156 of *LNCS*, pages 119–132. Springer, August 2004.

[56] T. Güneysu and C. Paar. Ultra high performance ECC over NIST primes on commercial FPGAs. In *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, volume 5154 of *LNCS*, pages 62–78. Springer, 2008.

[57] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[58] H. Hasse. Zur theorie der abstrakten elliptischen funktionenkörper i. die struktur der gruppe der divisorenklassen endlicher ordnung. *Journal für die reine und angewandte Mathematik*, 175:55–62, 1936.

[59] H. Heilbronn. On the average length of a class of finite continued fractions. In *Number Theory and Analysis*, pages 87–96. Springer, 1969.

[60] W. K. Jenkins and B. J. Leon. The use of residue number systems in the design of finite impulse response digital filters. *IEEE Transactions on Circuits and Systems*, 24(4):191–201, April 1977.

[61] M. Joye and J.-J. Quisquater. Hessian elliptic curves and side-channel attacks. In *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162 of *LNCS*, pages 402–410. Springer, May 2001.

[62] M. Joye and S.-M. Yen. The Montgomery powering ladder. In *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2523 of *LNCS*, pages 291–302. Springer, August 2002.

[63] A. Karatsuba and Y. Ofman. Multiplication of multi-digit numbers on automata. *Doklady Akad. Nauk SSSR*, 145(2):293–294, 1962. Translation in Soviet Physics-Doklady, 44(7), 1963, p. 595-596.

[64] S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-Rower architecture for fast parallel Montgomery multiplication. In *Proc. 19th International Conference on the Theory and Application of Cryptographic (EUROCRYPT)*, volume 1807 of *LNCS*, pages 523–538. Springer, May 2000.

[65] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In *Proc. 30th International Cryptology Conference (CRYPTO)*, volume 6223 of *LNCS*, pages 333–350. Springer, 2010.

[66] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 3rd edition, 1997.

[67] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[68] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. 16th International Cryptology Conference (CRYPTO)*, volume 1109 of *LNCS*, pages 104–113. Springer, August 1996.

[69] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proc. 19th International Cryptology Conference (CRYPTO)*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

[70] A. K. Lenstra and Lenstra H. Jr. (editors). *The development of the number field sieve*, volume 1554. Springer, 1993.

[71] Z. Lim and B. J. Phillips. An RNS-enhanced microprocessor implementation of public key cryptography. In *Proc. 41th Asilomar Conference on Signals, Systems and Computers*, pages 1430–1434. IEEE, November 2007.

[72] Z. Lim, B. J. Phillips, and M. Liebelt. Elliptic curve digital signature algorithm over GF(p) on a residue number system enabled microprocessor. In *Proc. IEEE Region 10 Conference (TENCON)*, pages 1–6, January 2009.

[73] P. Longa and C. Gebotys. Fast multibase methods and other several optimizations for elliptic curve scalar multiplication. In *Proc. 12th International Conference on Practice and Theory in Public Key Cryptography (PKC)*, volume 5443 of *LNCS*, pages 443–462. Springer, March 2009.

[74] Y. Ma, Z. Liu, W. Pan, and J. Jing. A high-speed elliptic curve cryptographic processor for generic curves over GF(p). In *Proc. 20th Selected Areas in Cryptography(SAC)*, LNCS, pages 421–437. Springer, 2013.

[75] E. Martín-López, A. Laing, T. Lawson, R. Alvarez, X.-Q. Zhou, and J. L. O'Brien. Experimental realization of Shor's quantum factoring algorithm using qubit recycling. *Nature Photonics*, 6(11):773–776, 2012.

[76] R. P. McEvoy, C. C. Murphy, W. P. Marnane, and M. Tunstall. Isolated WDDL: A hiding countermeasure for differential power analysis on FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 2(1):3, March 2009.

[77] A. J. Menezes, T. Okamoto, and S.A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39(5):1639–1646, Sep 1993.

[78] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[79] M. Mersenne. *Cogitata physico-mathematica , in quibus tam naturae quam artis effectus admirandi certissimis demonstrationibus explicantur*. A. Bertier Parisiis, 1644.

[80] V. Miller. Use of elliptic curves in cryptography. In *Proc. 5th International Cryptology Conference (CRYPTO)*, volume 218 of *LNCS*, pages 417–426. Springer, 1985.

[81] P. K. Mishra and V. Dimitrov. Efficient quintuple formulas for elliptic curves and efficient scalar multiplication using multibase number representation. In *Proc. 10th International Conference on Information Security (ISC)*, volume 4779 of *LNCS*, pages 390–406. Springer, October 2007.

[82] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[83] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *RAIRO Informatique Théorique et Applications*, 24(6):531–543, 1990.

[84] A. Moss, D. Page, and N. P. Smart. Toward acceleration of RSA using 3D graphics hardware. In *Proc. 11th IMA International Conference on Cryptography and Coding*, pages 364–383. Springer, December 2007.

[85] J.-M. Muller. *Arithmétique des ordinateurs*. Masson, 1989.

[86] B. Möller. Securing elliptic curve point multiplication against side-channel attacks. In *Information Security*, volume 2200 of *LNCS*, pages 324–334. Springer, 2001.

[87] H. Nozaki, M. Motoyama, A. Shimbo, and S. Kawamura. Implementation of RSA algorithm based on RNS Montgomery multiplication. In *Proc. 3rd Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162 of *LNCS*, pages 364–376. Springer, May 2001.

[88] National Security Agency/Central Security Service (NSA/CSS). Suite B implementer's guide to NIST SP 800-56A, 2009.

[89] National Security Agency/Central Security Service (NSA/CSS). Suite B implementer's guide to FIPS 186-3(ECDSA), 2010.

[90] National Institute of Standards and Technology (NIST). FIPS 197, advanced encryption standard (AES), 2001.

[91] National Institute of Standards and Technology (NIST). FIPS 186-4, digital signature standard (DSS), 2013.

[92] A. Omondi and B. Premkumar. *Residue number systems: theory and implementation*. Imperial College Press, 2007.

[93] B. Pascal. *Œuvres complètes*, chapter De Numeribus Multiplicibus, vol. 5, pages 117–128. Librarie Lefèvre, 1819.

[94] G. Perin, L. Imbert, L. Torres, and P. Maurine. Electromagnetic analysis on RSA algorithm based on RNS. In *Proc. 16th Euromicro Conference on Digital System Design (DSD)*, pages 345–352. IEEE, September 2013.

[95] B. J. Phillips, Y. Kong, and Z. Lim. Highly parallel modular multiplication in the residue number system using sum of residues reduction. *Applicable Algebra in Engineering, Communication and Computing*, 21(3):249–255, May 2010.

[96] S.C. Pohlig and M.E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, 24(1):106–110, Jan 1978.

[97] J.M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[98] K. C. Posch and R. Posch. Base extension using a convolution sum in residue number systems. *Computing*, 50(2):93–104, 1993.

[99] K. C. Posch and R. Posch. Modulo reduction in residue number systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):449–454, May 1995.

[100] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Proc. International Conference on Research in Smart Cards (E-smart)*, volume 2140 of *LNCS*, pages 200–210. Springer, September 2001.

[101] E. Raman, L. N. Chakrapani, K. Sankaranarayanan, and R. Parthasarathi. A scalable reconfigurable architecture for divisibility testing of variable long precision numbers. `http://www.cs.virginia.edu/~ks4kk/pubs/ekadhika_fccm02.pdf`. (personnal website from one author).

[102] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[103] J. Sakarovitch. *Elements of Automata Theory*, chapter Prologue: M. Pascal's Division Machine, pages 1–6. Cambridge, 2009.

[104] D. M. Schinianaki, A. P. Fournaris, H. E. Michail, A. P. Kakarountas, and T. Stouraitis. An RNS implementation of an $\mathbb{F}_p$ elliptic curve point multiplier. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(6):1202–1213, June 2009.

[105] D. Schinianakis and T. Stouraitis. A RNS montgomery multiplication architecture. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1167–1170, May 2011.

[106] D. Schinianakis and T. Stouraitis. An RNS modular multiplication algorithm. In *Proc. 20th International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 958–961. IEEE, Dec 2013.

[107] D. Schinianakis and T. Stouraitis. Multifunction residue architectures for cryptography. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(4):1156–1169, April 2014.

[108] D. Schinianakis and T. Stouraitis. An RNS barrett modular multiplication architecture. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2229–2232, June 2014.

[109] D.M. Schinianakis, A.P. Kakarountas, and T. Stouraitis. A new approach to elliptic curve cryptography: an RNS architecture. In *Proc. 13th IEEE Mediterranean Electrotechnical Conference (MELECON)*, pages 1241–1245, May 2006.

[110] I. Semaev. Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p. *Mathematics of Computation*, 67(221):353–356, 1998.

[111] O. Sentieys and A. Tisserand. Architecture reconfigurables FPGA. In *Technologies logicielles Architectures des systèmes*, volume H 1 196, pages 1–22. Techniques de l'Ingénieur, August 2012.

[112] A. P. Shenoy and R. Kumaresan. Fast base extension using a redundant modulus in RNS. *IEEE Transactions on Computers*, 38(2):292–297, February 1989.

[113] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proc. 35th Symposium on Foundations of Computer Science (FOCS)*, pages 124–134. IEEE, November 1994.

[114] N. P. Smart. The discrete logarithm problem on elliptic curves of trace one. *Journal of Cryptology*, 12(3):193–196, 1999.

[115] M. Soderstrand, W. K. Jenkins, G. Jullien, and F. Taylor, editors. *Residue Number System Arithmetic - Modern Applications in Digital Signal Processing*. IEEE, 1986.

[116] J. A. Solinas. Generalized mersenne numbers. Technical report, CORR-99-39 Center for Applied Cryptographic Research, University of Waterloo, 1999.

[117] J. Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1(3):397–405, February 1967.

[118] D. Suzuki. How to maximize the potential of FPGA resources for modular exponentiation. In *Proc. 9th Cryptographic Hardware and Embedded Systems (CHES)*, volume 4727 of *LNCS*, pages 272–288. Springer, September 2007.

[119] A. Svoboda and M. Valach. Operátorové obvody (operator circuits in czech). *Stroje na Zpracování Informací (Information Processing Machines)*, 3:247–296, 1955.

[120] N. S. Szabo and R. I. Tanaka. *Residue arithmetic and its applications to computer technology*. McGraw-Hill, 1967.

[121] R. Szerwinski and T. Guneysu. Exploiting the power of GPUs for asymmetric cryptography. In *Proc. 10th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 5154 of *LNCS*, pages 79–99. Springer, August 2008.

[122] K. Tiri and I. Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Proc. Conference on Design, Automation and Test in Europe (DATE) - Volume 1*, page 10246. IEEE, 2004.

[123] A.S. Wander, N. Gura, H. Eberle, V. Gupta, and S.C. Shantz. Energy analysis of public-key cryptography for wireless sensor networks. In *Proc. 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 324–328. IEEE, March 2005.

[124] H. S. Warren. *Hacker's Delight*. Addison-Wesley, 2003.

[125] A. C.-C. Yao. On the evaluation of powers. *SIAM Journal on Computing*, 5(1):100–103, 1976.

[126] G. Yao, J. Fan, R. Cheung, and I. Verbauwhede. Novel RNS parameter selection for fast modular multiplication. *IEEE Transactions on Computers*, 63(8):2099–2105, Aug 2014.

[127] G. X. Yao, J. Fan, R. C. C. Cheung, and I. Verbauwhede. Faster pairing coprocessor architecture. In *Proc. 5th Pairing-Based Cryptography (Pairing)*, volume 7708 of *LNCS*, pages 160–176. Springer, May 2012.

[128] S.-M. Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, Sep 2000.

[129] S.-M. Yen, S. Kim, S. Lim, and S. Moon. A countermeasure against one physical cryptanalysis may benefit another attack. In *Proc. 3rd International Conference on Information Security and Cryptology (ICISC)*, volume 2288 of *LNCS*, pages 414–427. Springer, December 2001.

Étude théorique et implantation matérielle d'unités de calcul en représentation modulaire des nombres pour la cryptographie sur courbes elliptiques

Ces travaux de thèse portent sur l'accélération de calculs de la cryptographie sur courbes elliptiques (ECC) grâce à une représentation peu habituelle des nombres, appelée représentation modulaire des nombres (ou RNS pour *residue number system*). Après un état de l'art de l'utilisation du RNS en cryptographie, plusieurs nouveaux algorithmes RNS, plus rapides que ceux de l'état de l'art, sont présentés. Premièrement, nous avons proposé un nouvel algorithme d'inversion modulaire en RNS. Les performances de notre algorithme ont été validées via une implantation FPGA, résultant en une inversion modulaire 5 à 12 fois plus rapide que l'état de l'art, pour les paramètres cryptographiques testés. Deuxièmement, un algorithme de multiplication modulaire RNS a été proposé. Cet algorithme décompose les valeurs en entrée et les calculs, afin de pouvoir réutiliser certaines parties lorsque c'est possible, par exemple lors du calcul d'un carré. Il permet de réduire de près de 25 % le nombre de pré-calculs à stocker et jusqu'à 10 % le nombre de multiplications élémentaires pour certaines applications cryptographiques (p. ex. le logarithme discret). Un algorithme d'exponentiation reprenant les mêmes idées est aussi présenté, réduisant le nombre de multiplications élémentaires de 15 à 22 %, contre un surcoût en pré-calculs à stocker. Troisièmement, un autre algorithme de multiplication modulaire RNS est proposé, ne nécessitant qu'une seule base RNS au lieu de 2 pour l'état de l'art, et utilisable uniquement dans le cadre ECC. Cet algorithme permet, pour certains corps bien spécifiques, de diviser par 2 le nombre de multiplications élémentaires et par 4 les pré-calculs à stocker. Les premiers résultats FPGA donnent des implantations de notre algorithme jusqu'à 2 fois plus petites que celles de l'algorithme de l'état de l'art, pour un surcoût en temps d'au plus 10 %. Finalement, une méthode permettant des tests de divisibilités multiples rapides est proposée, pouvant être utilisée en matériel pour un recodage de scalaire, accélérant certains calculs pour ECC.

Theoretical Study and Hardware Implementation of Arithmetical Units in Residue Number System (RNS) for Elliptic Curve Cryptography

The main objective of this PhD thesis is to speedup elliptic curve cryptography (ECC) computations, using the residue number system (RNS). A state-of-art of RNS for cryptographic computations is presented. Then, several new RNS algorithms, faster than state-of-art ones, are proposed. First, a new RNS modular inversion algorithm is presented. This algorithm leads to implementations from 5 to 12 times faster than state-of-art ones, for the standard cryptographic parameters evaluated. Second, a new algorithm for RNS modular multiplication is proposed. In this algorithm, computations are split into independant parts, which can be reused in some computations when operands are reused, for instance to perform a square. It reduces the number of precomputations by 25 % and the number of elementary multiplications up to 10 %, for some cryptographic applications (for example with the discrete logarithm). Using the same idea, an exponentiation algorithm is also proposed. It reduces from 15 % to 22 % the number of elementary multiplications, but requires more precomputations than state-of-art. Third, another modular multiplication algorithm is presented, requiring only one RNS base, instead of 2 for the state-of-art. This algorithm can be used for ECC and well-chosen fields, it divides by 2 the number of elementary multiplications, and by 4 the number of precomputations to store. Partial FPGA implementations of our algorithm halves the area, for a computation time overhead of, at worse, 10 %, compared to state-of-art algorithms. Finally, a method for fast multiple divisibility tests is presented, which can be used in hardware for scalar recoding to accelerate some ECC computations.